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)

October 9, 2013

A simple problem in Coq

$$\frac{\begin{array}{c} x\ y\ z : A \\ s : \text{list}\ A \end{array}}{x \in \text{append}\ s\ (z :: x :: y :: [])}$$

A simple problem in Coq

$$\frac{\begin{array}{r} x\ y\ z{:}A \\ s{:}\mathrm{list}\ A \end{array}}{x \in \mathrm{append}\ s\ (z :: x :: y :: [])}$$

Proof.

A simple problem in Coq

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

$$\frac{\begin{array}{c} x\ y\ z{:}A \\ s{:}\text{list } A \end{array}}{x \in \text{append } s\ (z :: x :: y :: [])}$$

Proof.

apply: inR.

A simple problem in Coq

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

$$\frac{\begin{array}{c} x\ y\ z{:}A \\ s{:}\text{list } A \end{array}}{x \in (z :: x :: y :: [])}$$

Proof.
apply: inR.

A simple problem in Coq

$$\frac{\begin{array}{c} x\ y\ z{:}A \\ s{:}\text{list } A \end{array}}{x \in (z :: x :: y :: [])}$$

Proof.
apply: inR.

A simple problem in Coq

$$\frac{\begin{array}{c} x\ y\ z{:}A \\ s{:}\mathrm{list}\ A \end{array}}{x \in (z :: x :: y :: [])}$$

Proof.
apply: inR.

A simple problem in Coq

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

$$\frac{\begin{array}{c} x\ y\ z{:}A \\ s{:}\text{list } A \end{array}}{x \in (z :: x :: y :: [])}$$

Proof.
apply: inR.
apply: in_tail.

A simple problem in Coq

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

$$\frac{\begin{array}{c} x\ y\ z{:}A \\ s{:}\text{list}\ A \end{array}}{x \in (x :: y :: [])}$$

Proof.

apply: inR.

apply: in_tail.

A simple problem in Coq

$$\frac{\begin{array}{c} x\ y\ z{:}A \\ s{:}\text{list } A \end{array}}{x \in (x :: y :: [])}$$

Proof.
apply: inR.
apply: in_tail.

A simple problem in Coq

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

$$\frac{\begin{array}{c} x\ y\ z{:}A \\ s{:}\text{list } A \end{array}}{x \in (x :: y :: [])}$$

Proof.

apply: inR.

apply: in_tail.

apply: in_head.

A simple problem in Coq

$$\frac{\begin{array}{c} x\ y\ z{:}A \\ s{:}\text{list } A \end{array}}{x \in (x :: y :: [])}$$

□

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

This script is:
Boring.

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

□

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

A simple problem in Coq

This script is:
Boring.
Fragile.

$$\frac{\begin{array}{c} x\ y\ z{:}A \\ s{:}\text{list } A \end{array}}{x \in (x :: y :: [])}$$

□

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

A simple problem in Coq

This script is:

Boring.

Fragile.

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

□

Proof.

apply: inR.

apply: in_tail.

apply: in_head.

Qed.

```
Ltac search x s :=
    match s with
    | append ?l ?r ⇒ let p := search x l in
                         constr: (inL r p)
    | append ?l ?r ⇒ let p := search x r in
                         constr: (inR l p)
    | (x :: ?s') ⇒ constr: (in_head x s)
    | (?y :: ?s') ⇒ let p := search x s' in
                         constr: (in_tail y x s' p)
    end.
```

search in Ltac

Logic of Coq includes a funct. language

```
Ltac search x s :=
    match s with
    | append ?l ?r ⇒ let p := search x l in
                         constr: (inL r p)
    | append ?l ?r ⇒ let p := search x r in
                         constr: (inR l p)
    | (x :: ?s') ⇒ constr: (in_head x s)
    | (?y :: ?s') ⇒ let p := search x s' in
                      constr: (in_tail y x s' p)
    end.
```

Logic of Coq includes a pure funct. language

```
Ltac search x s :=
    match s with
    | append ?l ?r ⇒ let p := search x l in
                          constr: (inL r p)
    | append ?l ?r ⇒ let p := search x r in
                          constr: (inR l p)
    | (x :: ?s') ⇒ constr: (in_head x s)
    | (?y :: ?s') ⇒ let p := search x s' in
                    constr: (in_tail y x s' p)
    end.
```

search in Ltac

Logic of Coq includes a pure funct. language
search requires impure features

```
Ltac search x s :=
    match s with
    | append ?l ?r ⇒ let p := search x l in
                         constr: (inL r p)
    | append ?l ?r ⇒ let p := search x r in
                         constr: (inR l p)
    | (x :: ?s') ⇒ constr: (in_head x s)
    | (?y :: ?s') ⇒ let p := search x s' in
                      constr: (in_tail y x s' p)
    end.
```

search in Ltac

#1: Syntax inspection

```
Ltac search x s :=
    match s with
    | append ?l ?r ⇒ let p := search x l in
                         constr: (inL r p)
    | append ?l ?r ⇒ let p := search x r in
                         constr: (inR l p)
    | (x :: ?s′) ⇒ constr: (in_head x s)
    | (?y :: ?s′) ⇒ let p := search x s′ in
                      constr: (in_tail y x s′ p)
    end.
```

search in Ltac

#2: Unbounded recursion

```
Ltac search x s :=
    match s with
    | append ?l ?r ⇒ let p := search x l in
                            constr: (inL r p)
    | append ?l ?r ⇒ let p := search x r in
                            constr: (inR l p)
    | (x :: ?s′) ⇒ constr: (in_head x s)
    | (?y :: ?s′) ⇒ let p := search x s′ in
                        constr: (in_tail y x s′ p)
    end.
```

search in Ltac

#3: Control flow (backtracking)

```
Ltac search x s :=
    match s with
    | append ?l ?r ⇒ let p := search x l in
                          constr: (inL r p)
    | append ?l ?r ⇒ let p := search x r in
                          constr: (inR l p)
    | (x :: ?s′) ⇒ constr: (in_head x s)
    | (?y :: ?s′) ⇒ let p := search x s′ in
                       constr: (in_tail y x s′ p)
    end.
```

Untyped!

```
Ltac search x s :=
    match s with
    | append ?l ?r ⇒ let p := search x l in
                       constr: (inL r p)
    | append ?l ?r ⇒ let p := search x r in
                       constr: (inR l p)
    | (x :: ?s') ⇒ constr: (in_head x s)
    | (?y :: ?s') ⇒ let p := search x s' in
                     constr: (in_tail y x s' p)
    end.
```

search in Ltac

Untyped!

No spec: search :???

```
Ltac search x s :=
    match s with
    | append ?l ?r ⇒ let p := search x l in
                        constr: (inL r p)
    | append ?l ?r ⇒ let p := search x r in
                        constr: (inR l p)
    | (x :: ?s') ⇒ constr: (in_head x s)
    | (?y :: ?s') ⇒ let p := search x s' in
                    constr: (in_tail y x s' p)
    end.
```

search in Ltac

Untyped!

Hard to maintain

```
Ltac search x s :=
    match s with
    | append ?l ?r ⇒ let p := search x l in
                        constr: (inL r p)
    | append ?l ?r ⇒ let p := search x r in
                        constr: (inR l p)
    | (x :: ?s') ⇒ constr: (in_head x s)
    | (?y :: ?s') ⇒ let p := search x s' in
                     constr: (in_tail y x s' p)
    end.
```

Untyped!

Hard to maintain

```
Ltac search x s :=
    match s with
    | append ?l ?r ⇒ let p := search x l in
                        constr: (inL r p)
    | append ?l ?r ⇒ let p := search x r in
                        constr: (inR l p)
    | (x :: ?s′) ⇒ constr: (in_head x s)
    | (?y :: ?s′) ⇒ let p := search x s′ in
                    constr: (in_tail y x s′ p)
    end.
```

search in Ltac

Untyped!

Hard to maintain

```
Ltac search x s :=
    match s with
    | append ?l ?r ⇒ let p := search x l in
                         constr: (inL r p)
    | append ?l ?r ⇒ let p := search x r in
                         constr: (inR l p)
    | (x :: ?s') ⇒ constr: (in_head x s) : x ∈ x :: s
    | (?y :: ?s') ⇒ let p := search x s' in
                      constr: (in_tail y x s' p)
    end.
```

search in Ltac

Untyped!

Debugging hell!

```
Ltac search x s :=
    match s with
    | append ?l ?r ⇒ let p := search x l in
                        constr: (inL r p)
    | append ?l ?r ⇒ let p := search x r in
                        constr: (inR l p)
    | (x :: ?s′) ⇒ constr: (in_head x s) : x ∈ x :: s
    | (?y :: ?s′) ⇒ let p := search x s′ in
                     constr: (in_tail y x s′ p)
    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 $\mathsf{M}\ A$.

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

Specification (type) should be:

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

search in Mtac

```
Definition search (x : A) :=
    mfix f (s : list A) : M (x ∈ s) :=
    mmatch s with
    | [l r] append l r ⇒ mtry  p ← f l;
                                ret (inL r p)
                          with _ ⇒ p ← f r;
                                ret (inR l p)
    | [s′] (x :: s′) ⇒ ret (in_head x s )
    | [y s′] (y :: s′) ⇒ p ← f s′; ret (in_tail y _ _ p)
    | _ ⇒ raise NotFound
    end.
```

search in Mtac

#1: Syntax inspection

```
Definition search (x : A) :=
    mfix f (s : list A) : M (x ∈ s) :=
    mmatch s with
    | [l r] append l r ⇒ mtry  p ← f l;
                               ret (inL r p)
                          with _ ⇒ p ← f r;
                               ret (inR l p)
    | [s'] (x :: s') ⇒ ret (in_head x s)
    | [y s'] (y :: s') ⇒ p ← f s'; ret (in_tail y _ _ p)
    | _ ⇒ raise NotFound
    end.
```

#2: Unbounded recursion

```
Definition search (x : A) :=
    mfix f (s : list A) : M (x ∈ s) :=
    mmatch s with
    | [l r] append l r ⇒ mtry  p ← f l;
                               ret (inL r p)
                          with _ ⇒ p ← f r;
                               ret (inR l p)
    | [s'] (x :: s') ⇒ ret (in_head x s)
    | [y s'] (y :: s') ⇒ p ← f s'; ret (in_tail y _ _ p)
    | _ ⇒ raise NotFound
    end.
```

search in Mtac

#3: Control flow (exceptions)

```
Definition search (x : A) :=
    mfix f (s : list A) : M (x ∈ s) :=
    mmatch s with
    | [l r] append l r ⇒ mtry  p ← f l;
                              ret (inL r p)
                          with _ ⇒ p ← f r;
                              ret (inR l p)
    | [s′] (x :: s′) ⇒ ret (in_head x s)
    | [y s′] (y :: s′) ⇒ p ← f s′; ret (in_tail y _ _ p)
    | _ ⇒ raise NotFound
    end.
```

search in Mtac

```
Definition search (x : A) :=
    mfix f (s : list A) : M (x ∈ s) :=
    mmatch s with
    | [l r] append l r ⇒ mtry  p ← f l;
                                ret (inL r p)
                            with _ ⇒ p ← f r;
                                ret (inR l p)
    | [s′] (x :: s′) ⇒ ret (in_head x s)
    | [y s′] (y :: s′) ⇒ p ← f s′; ret (in_tail y _ _ p)
    | _ ⇒ raise NotFound
    end.
```

Typed!

```
Definition search (x : A) :=
    mfix f (s : list A) : M (x ∈ s) :=
    mmatch s with
    | [l r] append l r ⇒ mtry  p ← f l;
                                ret (inL r p)
                           with _ ⇒ p ← f r;
                                ret (inR l p)
    | [s′] (x :: s′) ⇒ ret (in_head x s)
    | [y s′] (y :: s′) ⇒ p ← f s′; ret (in_tail y _ _ p)
    | _ ⇒ raise NotFound
    end.
```

Typed!

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

```
Definition search (x : A) :=
    mfix f (s : list A) : M (x ∈ s) :=
    mmatch s with
    | [l r] append l r ⇒ mtry  p ← f l;
                               ret (inL r p)
                           with _ ⇒ p ← f r;
                               ret (inR l p)
    | [s'] (x :: s') ⇒ ret (in_head x s)
    | [y s'] (y :: s') ⇒ p ← f s'; ret (in_tail y _ _ p)
    | _ ⇒ raise NotFound
    end.
```

search in Mtac

Typed!

Type checker

(1) **Definition** search $(x : A)$:=

(2) **mfix** f $(s : \text{list } A) : \text{M } (x \in s)$:=

(3) **mmatch** s **with**

(4) | $[l\ r]$ append $l\ r \Rightarrow$ **mtry** $p \leftarrow f\ l$;

(5) **ret** (inL r p)

(6) **with** _ $\Rightarrow p \leftarrow f\ r$;

(7) **ret** (inR l p)

(8) | $[s']$ $(x :: s') \Rightarrow$ **ret** (in_head x s)

(9) | $[y\ s']$ $(y :: s') \Rightarrow p \leftarrow f\ s'$; **ret** (in_tail y _ _ p)

(10) | _ $\Rightarrow$ **raise** NotFound

(11) **end**.

search in Mtac

Typed!

Type checker

```
Definition search (x : A) :=
    mfix f (s : list A) : M (x ∈ s) :=
    mmatch s with
    | [l r] append l r ⇒ mtry  p ← f l;
                                ret (inL r p)
                            with _ ⇒ p ← f r;
                                ret (inR l p)
    | [s'] (x :: s') ⇒ ret (in_head x s) : x ∈ x :: s
    | [y s'] (y :: s') ⇒ p ← f s'; ret (in_tail y _ _ p)
    | _ ⇒ raise NotFound
    end.
```

≠

search in Mtac

Typed!

Type inference

```
Definition search (x : A) :=
    mfix f (s : list A) : M (x ∈ s) :=
    mmatch s with
    | [l r] append l r ⇒ mtry  p ← f l;
                                ret (inL r p)
                           with _ ⇒ p ← f r;
                                ret (inR l p)
    | [s'] (x :: s') ⇒ ret (in_head _ _)
    | [y s'] (y :: s') ⇒ p ← f s'; ret (in_tail y _ _ p)
    | _ ⇒ raise NotFound
    end.
```

Typed!

Interactive programming

```
Definition search (x : A) :=
    mfix f (s : list A) : M (x ∈ s) :=
    mmatch s with
    | [l r] append l r ⇒ mtry  p ← f l;
                               ret (inL r p)
                          with _ ⇒ p ← f r;
                               ret (inR l p)
    | [s′] (x :: s′) ⇒ ret (in_head _ _)
    | [y s′] (y :: s′) ⇒ p ← f s′; ret (in_tail y _ _ p)
    | _ ⇒ raise NotFound
    end.
```

Typed!

Interactive programming

```
Program Definition search (x : A) :=
    mfix f (s : list A) : M (x ∈ s) :=
    mmatch s with
    | [l r] append l r ⇒ mtry  p ← f l;
                                ret _
                           with _ ⇒ p ← f r;
                                ret _
    | [s′] (x :: s′) ⇒ ret (in_head _ _)
    | [y s′] (y :: s′) ⇒ p ← f s′; ret (in_tail y _ _ p)
    | _ ⇒ raise NotFound
    end.
```

A simple problem in Coq, with Mtac

$$\frac{\begin{array}{c} x\ y\ z : A \\ s : \text{list } A \end{array}}{x \in \text{append } s\ (z :: x :: y :: [])}$$

Proof.

A simple problem in Coq, with Mtac

$$\frac{\begin{array}{c} x\ y\ z : A \\ s : \text{list}\ A \end{array}}{x \in \text{append}\ s\ (z :: x :: y :: [])}$$

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

A simple problem in Coq, with Mtac

$$\frac{\begin{array}{c} x\ y\ z : A \\ s : \text{list}\ A \end{array}}{x \in \text{append}\ s\ (z :: y :: x :: [])}$$

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

A simple problem in Coq, with Mtac

$$\frac{\begin{array}{c} x\ y\ z : A \\ s : \text{list } A \end{array}}{x \in \text{append } s\ (z :: y :: x :: [])}$$

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

run is a first-class citizen

A simple problem in Coq, with Mtac

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

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

run is a first-class citizen

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

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 $L : a \in s \rightarrow P\ a$
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!

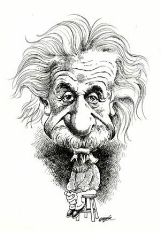
$$
\begin{aligned}
&\textbf{Inductive}\ \mathrm{M} : \mathrm{Type} \to \mathrm{Type} := \\
&\quad \mid \mathrm{ret} : A \to \mathrm{M}\ A \\
&\quad \mid \mathrm{bind} : \mathrm{M}\ A \to (A \to \mathrm{M}\ B) \to \mathrm{M}\ B \\
&\quad \mid \mathrm{mtry} : \mathrm{M}\ A \to (\mathrm{Exception} \to \mathrm{M}\ A) \to \mathrm{M}\ A \\
&\quad \mid \mathrm{raise} : \mathrm{Exception} \to \mathrm{M}\ A \\
&\quad \mid \mathrm{mfix} : ((\forall x : A.\ \mathrm{M}\ (B\ x)) \to (\forall x : A.\ \mathrm{M}\ (B\ x))) \\
&\qquad\qquad \to \forall x : A.\ \mathrm{M}\ (B\ x) \\
&\quad \ldots
\end{aligned}
$$

Mtac: tactic execution

Key idea #2:

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

Proof dev



Type inference

Kernel

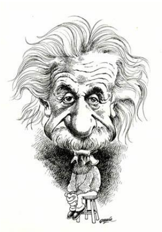
Mtac: tactic execution

Key idea #2:

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

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

Proof dev



Type inference

Kernel

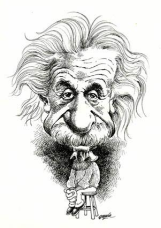
Mtac: tactic execution

Key idea #2:

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

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

Proof dev



Type inference

Kernel

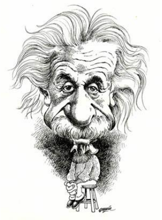
Mtac: tactic execution

Key idea #2:

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

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

Proof dev



Type inference

Kernel

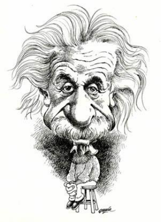
Mtac: tactic execution

Key idea #2:

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

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

Proof dev



Type inference

Kernel

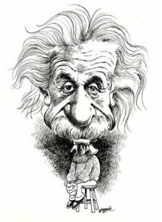
Mtac: tactic execution

Key idea #2:

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

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

Proof dev



Type inference

Kernel

Mtac: tactic execution

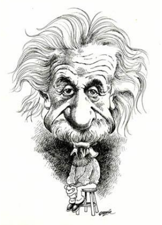
Key idea #2:

- 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

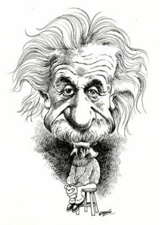
Key idea #2:

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

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

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

Proof dev



Type inference

Kernel

Mtac: tactic execution

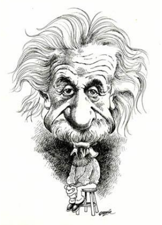
Key idea #2:

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

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

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

Proof dev



Type inference

Kernel

Mtac: tactic execution

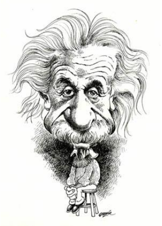
Key idea #2:

- 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

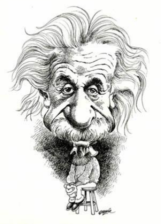
Key idea #2:

- 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

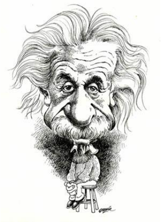
Key idea #2:

- 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

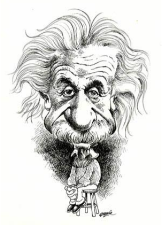
Key idea #2:

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

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

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

Proof dev



Type inference

Kernel

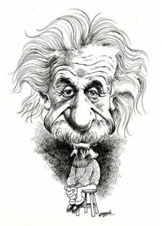
Key idea #2:

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

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

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

Proof dev



Type inference

Kernel

Mtac: tactic execution

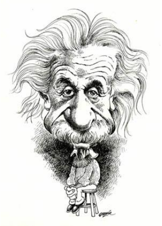
Key idea #2:

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

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

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

Proof dev



Type inference

Kernel

Mtac: tactic execution

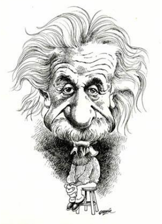
Key idea #2:

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

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

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

Proof dev



Type inference

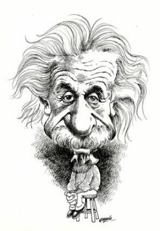
Kernel

Mtac: tactic execution

Key idea #2:

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

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



Type inference

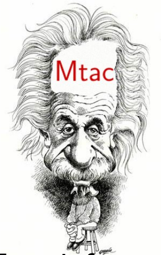
Kernel unmodified

Mtac: tactic execution

Key idea #2:

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

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



Type inference

Kernel unmodified

Related work

(Some) other typed tactic languages:

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

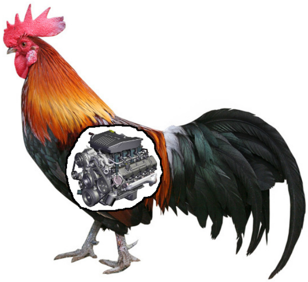
  Stronger typechecker, difficult to incorporate into Coq.

- 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/