Static and User-Extensible Proof Checking

Antonis Stampoulis    Zhong Shao
Yale University

POPL 2012
Proof assistants are becoming popular in our community

— CompCert [Leroy et al.]
— seL4 [Klein et al.]
— Four-color theorem [Gonthier et al.]

... but they’re still hard to use

— 1 to 1.5 weeks per paper proof page
— 4 pages of formal proof per 1 page of paper proof

[Asperti and Coen ‘10]
Formal proofs
— communicated to a fixed proof checker
— must spell out all details
— use domain-specific lemmas

Informal proofs
— communicated to a person
— rely on domain-specific intuition
— use “obviously”
Formal proofs
— communicated to a **fixed** proof checker
— must spell out all details
— use domain-specific lemmas

Informal proofs
— communicated to a person
— rely on **domain-specific intuition**
— use “obviously”

- calculus
- linear algebra
- arithmetic
We need tactics to omit details

— procedures that produce proofs
— domain-specific tactics good in large developments
— but difficult to write!
We need tactics to omit details

— procedures that produce proofs
— domain-specific tactics in large developments
— but difficult to write!

- untyped
- proofs within tactics can be wrong!
Proof assistants are hard to use because

1. cannot extend proof checker $\rightarrow$ lots of details
2. no checking for tactics $\rightarrow$ lots of potential errors

These are architectural issues
Our contribution:
A new architecture for proof assistants

1. cannot extend proof checker → lots of details
2. no checking for tactics → lots of potential errors
Our contribution:
A new architecture for proof assistants

1. extensible proof checker → omit lots of details
2. no checking for tactics → lots of potential errors
Our contribution: a new architecture for proof assistants.

1. Extensible proof checker → omit lots of details
2. No checking for tactics → lots of potential errors

Full programming model soundness guaranteed.
Our contribution: A new architecture for proof assistants

1. extensible proof checker → omit lots of details
   1. cannot extend proof checker → lots of details

2. no checking for tactics → lots of potential errors
   2. extensible checking for tactics → lots of errors avoided
Our contribution:
A new architecture for proof assistants

1. extensible proof checker \rightarrow omit lots of details
2. no checking for tactics \rightarrow lots of potential errors

static checking of contained proofs
Our contribution:
A new architecture for proof assistants

1. extensible proof checker \(\rightarrow\) omit lots of details
2. no checking for tactics \(\rightarrow\) lots of potential errors

1. cannot extend proof checker \(\rightarrow\) lots of details
2. extensible checking for tactics \(\rightarrow\) lots of errors avoided
Our contribution:

A new architecture for proof assistants

1. Extensible proof checker → omit lots of details
2. Extensible checking for tactics → lots of errors avoided
Our contribution:
A new architecture for proof assistants

1. extensible proof checker → omit lots of details
2. no checking for tactics → lots of potential errors

More specifically:
- a new language design
- a new implementation
- and a new metatheory

based on VeriML [ICFP’10]
Architecture of proof assistants
## Architecture of proof assistants: main notions

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proof object</td>
<td>Derivation in a logic</td>
</tr>
<tr>
<td>Proof checker</td>
<td>Checks proof objects</td>
</tr>
<tr>
<td>Tactic</td>
<td>Function producing proof objects</td>
</tr>
<tr>
<td>Proof script</td>
<td>Combination of tactics; program producing a proof object</td>
</tr>
</tbody>
</table>
Architecture of proof assistants: Checking proof objects

Proof object

Proof checker
Architecture of proof assistants: Checking proof objects

Proof object

Proof checker

Conversion rule

Implicitly check equivalences (proof omitted)
Architecture of proof assistants: Checking proof objects

Proof object

Proof checker

Coq

βη-conversion
Architecture of proof assistants: Checking proof objects

Proof object

Proof checker

Conversion

CoqMT

$\beta\iota$-conversion + linear arithmetic
Architecture of proof assistants: Checking proof objects

Proof object
- rich static information
- (de)composable
- checking not extensible
Architecture of proof assistants: Validating proof scripts

- extensible through tactics
- rich programming model
- no static information
- not (de)composable
- hidden proof state
Architecture of proof assistants: Validating proof scripts

- extensible through tactics
- rich programming model
- no static information
- not (de)composable
- hidden proof state

use conversion for more robust scripts
Moving to typed proof scripts
Moving to typed proof scripts

Proof script

- Arithmetic tactic
- User tactic
- Other tactic

Proof object

Proof checker

Conversion

Evaluation
Moving to typed proof scripts

Proof script

Proof object

Proof checker

Conversion

Arithmetic tactic
User tactic
Other tactic

evaluation

Proof checker

Conversion

evaluation
Moving to typed proof scripts

- Proof script
  - Arithmetic tactic
  - User tactic
  - Other tactic

- Proof object
  - Proof checker
  - Conversion

- Evaluation
  - Type checker
  - Proof checker
  - Conversion
  - Evaluation
Moving to typed proof scripts

Proof script
- Proof object
  - Proof checker
  - Conversion
  - Evaluation

Typed proof script
- Type checker
  - Proof checker
  - Conversion
  - Evaluation

Tactic: Arithmetic, User, Other

Type checker

Evaluation
Moving to typed proof scripts + extensible conversion

Typed proof script

Type checker

Proof checker

Conversion

evaluation

Arithmetic tactic
User tactic
Other tactic
Moving to typed proof scripts + extensible conversion

Key insight: conversion is just a hardcoded trusted tactic
Key insight: conversion is just a hardcoded trusted tactic but we can trust other tactics too if they have the right type.
Moving to typed proof scripts + extensible conversion

Key insight: conversion is just a hardcoded trusted tactic

but we can trust other tactics too if they have the right type

none of them needs to be hardcoded!

Typed proof script

Type checker

Proof checker

User-specified Conversion

evaluation

Other tactic
Typed proof scripts + extensible conversion

- rich static information
- user chooses conversion
- extensible static checking
- smaller proof checker
- can generate proof objects
Type checking tactics: an example

classificationCaseCheck : (P : Prop) → (P' : Prop) → 
(proof: P = P')

- check propositions for equivalence
- return a proof if they are
- raise an exception otherwise
Type checking tactics:
- check propositions for equivalence
- return a proof if they are
- raise an exception otherwise

Metatheory result 1. Type safety
If evaluation succeeds, the returned proof object is valid

conversionCheck : (P : Prop) → (P' : Prop) → (proof: P = P')
Type checking tactics:
- an example
  - check propositions for equivalence
  - return a proof if they are
  - raise an exception otherwise

**Metatheory result 1. Type safety**
If evaluation succeeds, the returned proof object is valid.

conversionCheck : (P : Prop) → (P' : Prop) →
(proof: P = P')

**Metatheory result 2. Proof erasure**
If evaluation succeeds, a valid proof object exists even if it’s not generated.
- check
- return
- raise an exception otherwise
Two modes of evaluation

- Typed proof script
  - Type checker
    - Proof checker
  - User-specified Conversion
Two modes of evaluation

Typed proof script

Type checker

Proof checker

User-specified Conversion

evaluation

proof object
Two modes of evaluation

Typed proof script

Type checker

Proof checker

User-specified Conversion

proof erasure evaluation

evaluation

proof object
Static checking = type checking + staging under proof-erasure
Static checking = type checking + staging under proof-erasure
Static checking = type checking + staging under proof-erasure
Static checking = type checking + staging under proof-erasure

Typed proof script

Type checker

Proof checker

User-specified conversion

typechecking

stage-one evaluation with proof erasure

evaluation of residual program

evaluation
A stack of conversion rules

- arithmetic simplification
- congruence closure
- $\beta$-conversion
A stack of conversion rules

- arithmetic simplification
- congruence closure
- βι-conversion

Conversion in Coq removed from trusted base
A stack of conversion rules

- arithmetic simplification
- congruence closure
- β₁-conversion

makes most uses of rewrite/autorewrite unnecessary
A stack of conversion rules

- arithmetic simplification
- congruence closure
- \(\beta\iota\)-conversion

ring\_simplify\ for\ Nat close to CoqMT
A stack of conversion rules

- arithmetic simplification
- congruence closure

- no additions to logic metatheory
- actually, with reductions
- no proof by reflection or translation validation
- leveraging static proof scripts
A stack of conversion rules

- arithmetic simplification
- congruence closure
- βι-conversion

1. arithmetic simplification
2. congruence closure
3. βι-conversion
A stack of conversion rules

- arithmetic simplification
- naïve arithmetic conversion
- congruence closure
- naïve equality conversion
- $\beta$-conversion
A stack of conversion rules

- arithmetic simplification
- naïve arithmetic conversion
- congruence closure
- naïve equality conversion
- $\beta_1$-conversion

- potentially non-terminating
- reduce proving for “real” versions
Require Import Arith.

Variable x : Nat.

Theorem test1 : 0 + x = x.
  trivial.
  Qed.

Theorem test2 : x + 0 = x.
  trivial.
  Qed.
Require Import Arith.
Variable x : Nat.

Theorem test1 : 0 + x = x.
  trivial.
  Qed.

Theorem test2 : x + 0 = x.
  trivial.
  Qed.

Proof completed
Require Import Arith.

Variable x : Nat.

Theorem test1 : 0 + x = x.
  trivial.
Qed.

Theorem test2 : x + 0 = x.
  trivial.
Qed.

Attempt to save an incomplete proof
Require Import Arith.
Variable x : Nat.

Theorem test1 : 0 + x = x.
  trivial.
Qed.

Theorem test2 : x + 0 = x.
  trivial.
Qed.

Conversion rule can prove this but can’t prove this
Let’s add this to our conversion rule!

\textbf{lemma1} : \forall x. x + 0 = x

\textbf{lemma2} : \forall xy. x + \text{(succ } y\text{)} = \text{succ}(x + y)

- write a rewriter based on these lemmas
- register it with conversion
- now it’s trivial; lemmas used implicitly
lemma1 : \( \forall x. x + 0 = x \)
lemma2 : \( \forall xy. x + (\text{succ } y) = \text{succ}(x + y) \)
lemma1 : \( \forall x . x + 0 = x \)

lemma2 : \( \forall xy . x + (\text{succ } y) = \text{succ}(x + y) \)

rewriter : \( (t : T) \rightarrow (t' : T \times \text{proof} : t = t') \)

rewriter \( t = \)

match \( t \) with

\( x + y \mapsto \)

let \( y' \), \( H \) = rewriter \( y \) in

match \( y' \) with

\( 0 \mapsto x, ? \)

| \( \text{succ } y'' \mapsto \text{succ } (x + y'') \), ? |

| \( - \mapsto t, ? \) |

| \( \cdots \) |
\textbf{lemma 1} : \ \forall x . x + 0 = x

\textbf{lemma 2} : \ \forall xy . x + (\text{succ} \ y) = \text{succ} (x + y)

\textbf{rewriter} : \ (t : T) \rightarrow (t' : T \times \text{proof} : t = t')

\textbf{rewriter} \ t =

\textbf{match} \ t \ \text{with}

\quad x + y \mapsto

\quad \text{let} \ y' , \ H = \text{rewriter} \ y' \ \text{in}

\quad \text{match} \ y' \ \text{with}

\quad \quad 0 \mapsto x , ?

\quad \quad \mid \text{succ} \ y'' \mapsto \text{succ} \ (x + y'') , ?

\quad \quad \mid \_ \mapsto t , ?

\quad \quad \ldots
**Lemma 1**: \( \forall x. x + 0 = x \)

**Lemma 2**: \( \forall xy. x + (\text{succ } y) = \text{succ}(x + y) \)

**Rewriter**: \( \begin{align*}
(t : T) &\rightarrow (t' : T \times \text{proof : } t = t') \\
\end{align*} \)

rewriter \( t = \)

match \( t \) with

\( x + y \mapsto \)

let \( y', H = \text{rewriter } y \)

match \( y' \) with

\( 0 \mapsto x, ? \)

\( \frac{x, y, y'' : T \quad H : y = \text{succ } y''}{\text{proof : } x + y = \text{succ} (x + y'')} \)

\( \text{lemma2 } x y'' \)
\[
\text{lemma 1} : \forall x . x + 0 = x \\
\text{lemma 2} : \forall xy . x + (\text{succ } y) = \text{succ}(x + y)
\]

rewriter : \( (t : T) \rightarrow (t' : T \times \text{proof} : t = t') \)

rewriter \( t = \)

match \( t \) with

\( x + y \mapsto \)

let \( y' , H = \text{rewriter } y \) /n
match \( y' \) with

\( 0 \mapsto x , ? \)

\( \mid \text{succ } y'' \mapsto \text{succ } (x + y'') , \)

\( \mid - \mapsto t , ? \)

\( \mid \ldots \)
\textbf{lemma 1}: \ \forall x. x + 0 = x \\
\textbf{lemma 2}: \ \forall xy. x + (\text{succ } y) = \text{succ}(x + y) \\

\textbf{rewriter}: \ (t : T) \rightarrow (t' : T \times \text{proof} : t = t') \\
\textbf{rewriter } t = \\
\text{match } t \text{ with} \\
\hspace{1cm} x + y \mapsto \\
\hspace{2cm} \text{let } y', H = \text{rewriter } y \text{ in} \\
\hspace{2cm} \text{match } y' \text{ with} \\
\hspace{3cm} 0 \mapsto x, ? \\
\hspace{3cm} \mid \text{succ } y'' \mapsto \text{succ } (x + y''), ? \\
\hspace{3cm} \mid \_ \mapsto t, ? \\
\hspace{1cm} \cdots
\textbf{lemma 1} : \ \forall x . x + 0 = x  
\textbf{lemma 2} : \ \forall xy . x + (\text{succ } y) = \text{succ}(x + y)  
\textbf{lemma 2'} : \ \forall xyy'. y = \text{succ } y' \rightarrow \  
\hspace{1cm} x + y = \text{succ}(x + y') = t'  
\textbf{rewriter } t =  
\text{match } t \text{ with }  
\hspace{1cm} x + y \mapsto  
\text{let } y', H = \text{rewriter } y' \text{ in }  
\text{match } y' \text{ with }  
\hspace{1cm} 0 \mapsto x, ?  
\hspace{1cm} | \text{succ } y'' \mapsto \text{succ } (x + y''),  
\hspace{1cm} | - \mapsto t, ?  
\hspace{1cm} | \cdots  
\textbf{proof : } x + y = \text{succ } (x + y'')  
\begin{align*}  
x, y, y'' : & T \\
H : & y = \text{succ } y''  \\
\hline  
\text{proof : } & x + y = \text{succ } (x + y'') 
\end{align*}  
\textbf{lemma 2' } x \ y \ y''
\textbf{Lemma 1} : \ \forall x. x + 0 = x \\
\textbf{Lemma 2} : \ \forall y. x + (\text{succ } y) = \text{succ} (x + y) \\
\textbf{Lemma 2'} : \ \forall y'. y = \text{succ } y' \rightarrow \\
\quad x + y = \text{succ} (x + y') = t' \\
\text{Rewriter } t = \\
\text{match } t \text{ with } \\
\quad x + y \rightarrow \\
\quad \text{let } y', H = \text{rewriter } y' \rightarrow \\
\text{match } y' \text{ with } \\
\quad 0 \rightarrow x, ? \\
\quad | \text{succ } y'' \rightarrow \text{succ} (x + y''), , 
\quad | - \rightarrow t, ? \\
\quad | \ldots
Lemma 1: \( \forall x. x + 0 = x \)

Lemma 2: \( \forall x y. x + (\text{succ } y) = \text{succ}(x + y) \)

Exact: \( (proof : P) \rightarrow (proof : P') \)

Rewriter: \( (t : T) \rightarrow (t' : T \times proof : t = t') \)

Rewriter \( t = \)

match \( t \) with

\[ x + y \mapsto \]

let \( y', H = \) rewriten \( y \)

match \( y' \) with

\[ 0 \mapsto x, ? \]

\[ \text{succ } y'' \mapsto \text{succ } (x + y''), ? \]

\[ - \mapsto t, ? \]

\[ \ldots \]
\textbf{Lemma 1} : \( \forall x. x + 0 = x \)

\textbf{Lemma 2} : \( \forall xy. x + (\text{succ } y) = \text{succ}(x + y) \)

\textbf{Exact} : \((\text{proof} : P) \rightarrow (\text{proof} : P')\)

\textbf{Rewriter} : \((t : T) \rightarrow (t' : T \times \text{proof} : t = t')\)

\textbf{Rewriter} \( t = \)

\textbf{match} \( t \) \textbf{with}

\( x + y \mapsto \)

\textbf{let} \( y', H = \text{rewriter} y' \)

\textbf{match} \( y' \) \textbf{with}

\( 0 \mapsto x, ? \)

\( \mid \text{succ } y'' \mapsto \text{succ } (x + y''), ? \)

\( \mid - \mapsto t, ? \)

\( \mid \ldots \)
lemmas:

1. \( \forall x. x + 0 = x \)
2. \( \forall xy. x + (\text{succ } y) = \text{succ}(x + y) \)

rewriter:

\( \text{let } y', H = \text{rewriter } y \quad \text{match } y' \text{ with} \)

- \( 0 \mapsto x, ? \)
- \( \text{succ } y'' \mapsto \text{succ } (x + y''), ? \)
- \( _{-} \mapsto t, ? \)

\( \Box \)

\( \text{proof : } t \mapsto t' \)
**Lemma 1**: $\forall x. x + 0 = x$

**Lemma 2**: $\forall x y. x + (\text{succ } y) = \text{succ}(x + y)$

**Exact**: $(\text{proof} : P) \rightarrow (\text{proof} : P')$

rewriter: $(t : T) \rightarrow (t' : T \times \text{proof} : t = t')$

rewriter $t =$

match $t$ with

$x + y$ $\mapsto$

let $y', H = \text{rewriter } y$ in

match $y'$ with

$0$ $\mapsto x, ?$

| succ $y''$ $\mapsto \text{succ } (x + y''),$
| _ $\mapsto t, ?$

| ...
\[ \text{lemma 1} : \ \forall x. x + 0 = x \]
\[ \text{lemma 2} : \ \forall xy. x + (\text{succ } y) = \text{succ}(x + y) \]

**Exact** : \((\text{proof} : P) \rightarrow (\text{proof} : P')\)

**rewriter** : \((t : T) \rightarrow (t' : T \times \text{proof} : t = t')\)

rewriter \(t = \)

match \(t\) with
\[ x + y \mapsto \]
let \(y', H = \text{rewriter } y\) 

- not checked statically
- recomputed many times

\[ \text{Exact} \quad (\text{lemma 2 } x \ y \ y') \]

\[ x, y, y'' : T \]
\[ H : y = \text{succ } y'' \]
\[ \text{proof} : x + y = \text{succ } (x + y'') \]
lemma1 : \( \forall x. x + 0 = x \)

lemma2 : \( \forall xy. x + (\text{succ } y) = \text{succ} (x + y) \)

Exact : \((\text{proof} : P) \rightarrow (\text{proof} : P')\)

rewriter : \((t : T) \rightarrow (t' : T \times \text{proof} : t = t')\)

rewriter \( t = \)
match \( t \) with
\[ x + y \mapsto \]
let \( y', H = \) rewriter \( y \)

match \( y' \) with
\[ 0 \mapsto x, ? \]
\[ \text{succ } y'' \mapsto \text{succ} (x + y''), ? \]
\[ _ \mapsto t, ? \]
\[ \ldots \]
lemma1 : $\forall x. x + 0 = x$

lemma2 : $\forall xy. x + (\text{succ } y) = \text{succ}(x + y)$

Exact : $(proof : P) \rightarrow (proof : P')$

rewriter : $(t : T) \rightarrow (t' : T \times proof : t = t')$

rewriter $t =$

match $t$ with

$x + y \mapsto$

let $y', H = \text{rewriter } y$

match $y'$ with

$0 \mapsto x, ?$

| succ $y'' \mapsto \text{succ}(x + y''), \{ \text{Exact} \ (\text{lemma2 } x y'') \}_{\text{static}}$

| __ \mapsto t, ?$

| __

| __
\[ \text{lemma 1} : \forall x. x + 0 = x \]
\[ \text{lemma 2} : \forall x y. x + (\text{succ } y) = \text{succ}(x + y) \]

**Exact**: \((\text{proof} : P) \rightarrow (\text{proof} : P')\)

**Rewriter**: \((t : T) \rightarrow (t' : T \times \text{proof} : t = t')\)

**Rewriter** \(t =\)

**match** \(t\) with

\[ x + y \mapsto \]

- checked at definition time
- computed once
- transformation of runtime arguments to constant arguments

\[ x', y, y'' : T \]
\[ H : y = \text{succ } y'' \]
\[ \quad \rightarrow \quad \]
\[ \text{proof} : x + y = \text{succ} (x + y'') \]

\[ \{ \text{Exact} \} \]
\[ \{ \text{lemma 2 } x \ y'' \} \_\text{static} \]
How does it work?

rewriter \( \Phi t = \)
match \( t \) with
\[
[\Phi] x + y \mapsto \\
\text{let } [\Phi] y', [\Phi] H = \text{rewriter } [\Phi] y \text{ in} \\
\text{match } y' \text{ with} \\
[\Phi] 0 \mapsto [\Phi] x, ? \\
| [\Phi] \text{succ } y'' \mapsto [\Phi] \text{succ } (x + y''), \left\{ \begin{array}{l}
\text{Exact} \\
(\text{lemma2 } x \ y'')
\end{array} \right\}_\text{static} \\
| _ \mapsto [\Phi] t, ? \\
| \ldots
\]
How does it work?

**Exact:** \((\Phi : \text{ctx}) \rightarrow (\text{proof} : [\Phi] P) \rightarrow (\text{proof} : [\Phi] P')\)

**Rewriter:**

\[(\Phi : \text{ctx}) \rightarrow (T : [\Phi] \text{Type}) \rightarrow (t : [\Phi] T) \rightarrow (t' : [\Phi] T \times \text{proof} : [\Phi] t = t')\]

**Rewriter \(\Phi t =\)**

**Match \(t\) with**

\(\Phi x + y \mapsto\)

let \([\Phi] y', [\Phi] H = \text{rewriter} [\Phi] y\) in

**Match \(y'\) with**

\((\Phi) 0 \leftrightarrow [\Phi] x, ?\)

| \([\Phi] \text{succ } y'' \leftrightarrow [\Phi] \text{succ } (x + y'')\), \(\left\{\begin{array}{l}
\text{Exact} \\
(\text{lemma2 } x y'')
\end{array}\right\}_{\text{static}}\)|

| \(- \leftrightarrow [\Phi] t, ?\)|

| \(\ldots\)|
\[\text{let static } pf = \]
\[\quad \text{let } \Phi' = [x, y, y'' : \text{Nat}, H : y = \text{succ } y''] \text{ in } \]
\[\quad \text{Exact } \Phi' (\left[\Phi'\right] \text{lemma2 } x \ y'') \]
\[\text{in } \]
\[\left[\Phi\right] pf / [x / \text{id}_\Phi, y / \text{id}_\Phi, y' / \text{id}_\Phi, H / \text{id}_\Phi] \]

rewriter : 
\[\Phi \ t = \]
\[\text{match } t \text{ with } \]
\[\quad \left[\Phi\right] x + y \mapsto \]
\[\quad \text{let } \left[\Phi\right] y', \left[\Phi\right] H = \text{rewriter } \left[\Phi\right] y \text{ in } \]
\[\quad \text{match } y' \text{ with } \]
\[\quad \quad \left[\Phi\right] 0 \mapsto \left[\Phi\right] x, ? \]
\[\quad \mid \left[\Phi\right] \text{succ } y'' \mapsto \left[\Phi\right] \text{succ } (x + y''), \{ \text{Exact } \left\{ \text{lemma2 } x \ y'' \right\} \}_{\text{static}} \]
\[\quad \mid _- \mapsto \left[\Phi\right] t, ? \]
\[\mid \ldots \]
letstatic $pf =$

let $\Phi' = [x, y, y' : \text{Nat}, H : y = \text{succ} \ y'']$ in

exact $\Phi'$ ([$\Phi'$] lemma2 $x \ y''$)

rewriter : [$\Phi$] $t =$

match $t$ with

[[$\Phi$] $x +$]

let

$H = \text{rewriter} \ [\Phi] \ y \ in$

\[
\begin{align*}
\bullet; \Sigma; \Gamma|_{\text{static}} \vdash e : \tau & \quad \Psi; \Sigma; \Gamma, x : s \tau \vdash e' : \tau \\
\overline{} & \quad \Psi; \Sigma; \Gamma \vdash \text{letstatic} \ x = e \ in \ e' : \tau
\end{align*}
\]

\[
\left\{\begin{array}{l}
\vdash \left[\Phi\right] t, \ ? \\
\vdots
\end{array}\right\}_{\text{static}}
\]
Implementation

http://www.cs.yale.edu/homes/stampoulis/

• type inferencing and implicit arguments
• compiler to OCaml
• rewriter code generation
• inductive types

Talk to me for a demo!
What’s in the paper and TR

• Static and dynamic semantics
• Metatheory:
  *Type-safety theorem*
  *Proof erasure theorem*
  *Static proof script transformation*
• Implementation details and examples implemented
• Typed proof scripts as flexible proof certificates
Related work

• proof-by-reflection
  – restricted programming model (total functions)
  – tedious to set up
  – here: no need for termination proofs

• automation through canonical structures / unification hints
  – restricted programming model (logic programming)
  – very hard to debug
Summary

• a new architecture for proof assistants
• user-extensible checking of proofs and tactics
• minimal trusted core
• reduce required effort for formal proofs

Thanks!

http://www.cs.yale.edu/homes/stampoulis/
Backup slides
Type checking proofs and tactics

- manipulate proofs and propositions in a type-safe manner
- dependent pattern matching on logical terms
- logic and computation are kept separate
- Beluga [Pientka & Dunfield ‘08]
- Delphin [Poswolsky & Schürmann ‘08]
- VeriML [Stampoulis & Shao ‘10]
Related work

• LCF family of proof assistants
  – *no information while writing proof scripts/tactics*

• Coq / CoqMT
  – *conversion rule is fixed*
  – *changing it requires re-engineering*

• NuPRL
  – *extensional type theory and sophisticated conversion*
  – *here: user decides conversion (level of undecidability)*

• Beluga / Delphin
  – *use as metalogic for LF*
  – *here: the logic is fixed; the language is the proof assistant*