A user-extensible and safe alternative to the conversion rule using VeriML

Antonis Stampoulis   Zhong Shao

Department of Computer Science, Yale University

TYPES 2011

Proof assistants are great!

- Verification of practical software
- Mathematical proofs
- Metatheory of programming languages

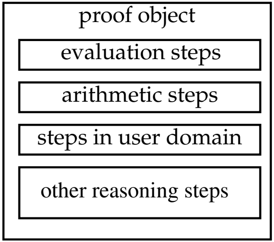
More work needed

- Foundational issues
  properties of the logic used as the foundation?
- Scalability
  how to scale to verification of 100k lines of code?
- Ease-of-use
  computer proofs closer to pen-and-paper ones?

Claim: architectural issues hurting all three aspects

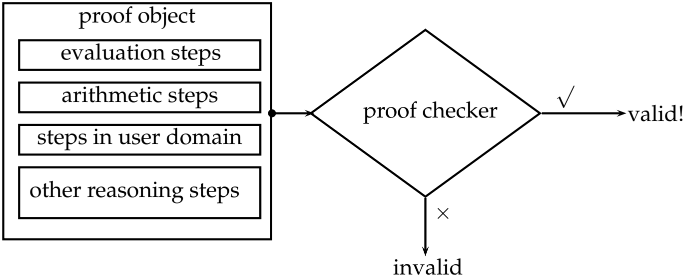
Proof object

A derivation inside a logic.



Proof checker

Program checking validity of proof objects.

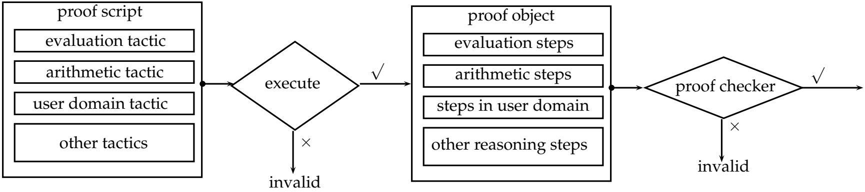


Tactics

Proof objects are very detailed, so we use tactics to produce parts of them.

Proof script

A program that produces a proof object by combining tactics.

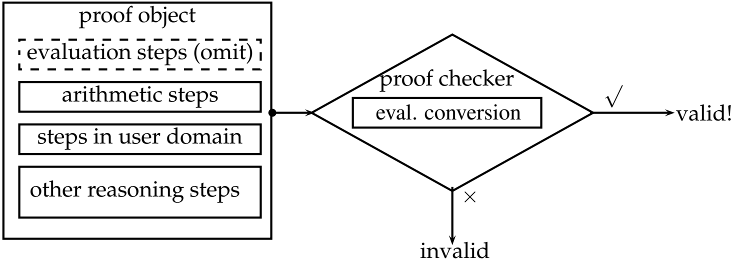


Proof assistant

A language environment to develop and execute proof scripts and tactics; along with a library of tactics and theorems.

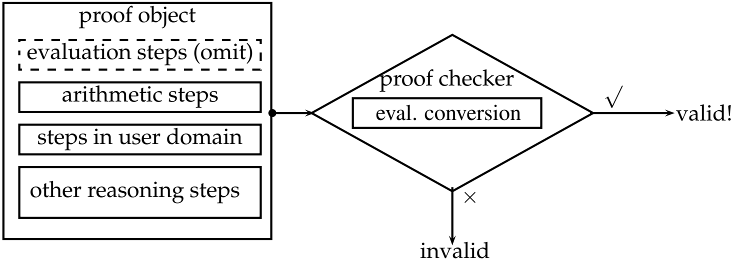
Checking proof objects

Checking proof objects



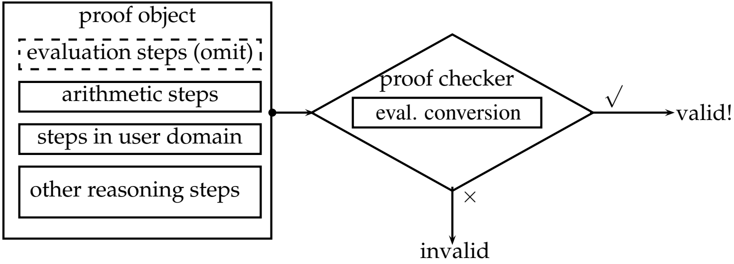
- to keep size manageable: conversion rule
- more sophisticated conversion → simpler proofs

Checking proof objects



- to keep size manageable: conversion rule
  - decides whether two propositions are equivalent
  - proof can be omitted
- more sophisticated conversion → simpler proofs

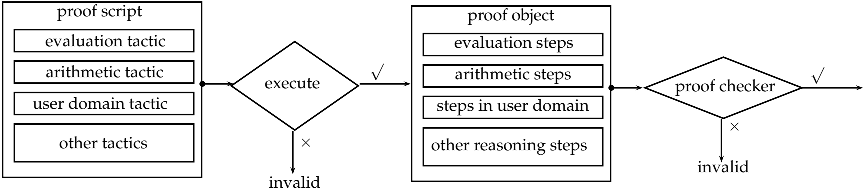
Checking proof objects



- to keep size manageable: conversion rule
- more sophisticated conversion → simpler proofs
  - e.g. CoqMT
  - but also larger trusted base
  - more complicated metatheory
  - cannot add user extensions

Checking proof scripts

Checking proof scripts



- validation: execute and check
- user-extensible: by writing tactics
- no static checking: completely untyped
- more robust by using proof objects

VeriML (ICFP 2010)

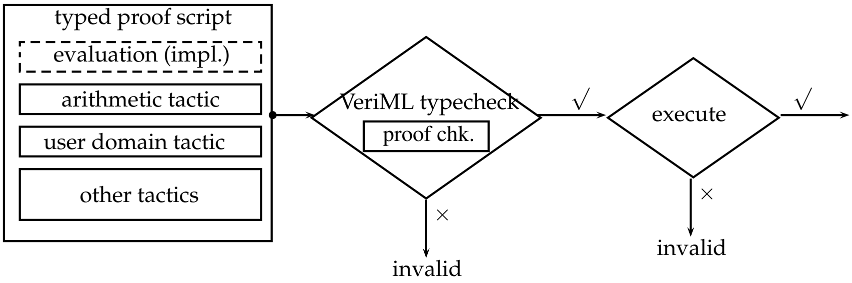
A language that supports dependently typed programming over logical terms.

VeriML (ICFP 2010)

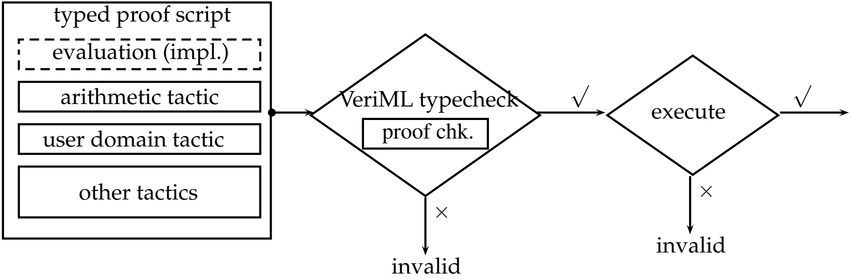
A language that supports dependently typed programming over logical terms.

Develop typed tactics and thus typed proof scripts

e.g. $\mathsf{auto} : (P : \mathsf{Prop}) \to \mathsf{option}\ \langle \mathsf{pf} : P \rangle$

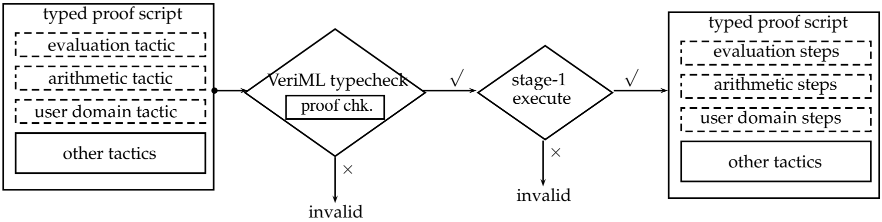


But still...



- proof checker still uses fixed conversion rule
- therefore static checking not user-extensible

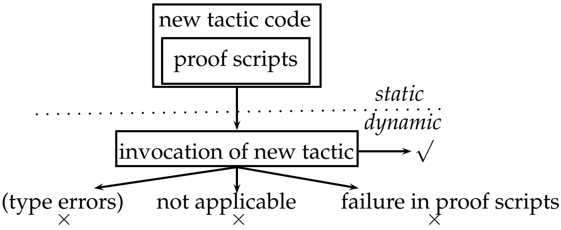
Extensible conversion rule



- user-defined extensions to the proof checker
- based on typed conversion tactics
- type-safety of VeriML guarantees soundness
- essentially, some tactics evaluated prior to others (simple staging)
- needs the extra type information

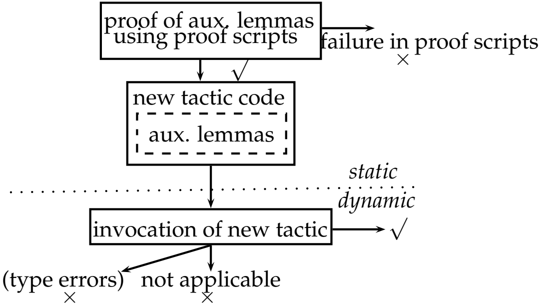
Checking tactics

Checking tactics: Coq, HOL



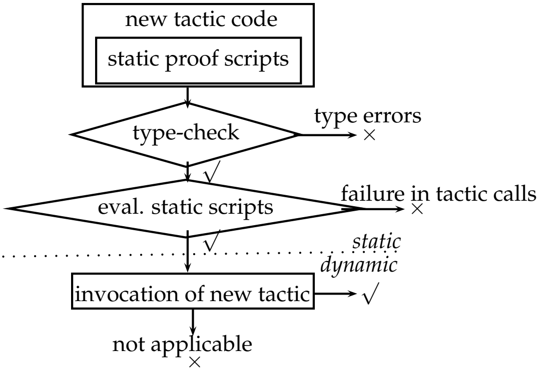
- tactics include proof scripts
- evaluated only upon invocation!
- validity of embedded proof scripts not known statically

Checking tactics: Coq, HOL (better engineering)



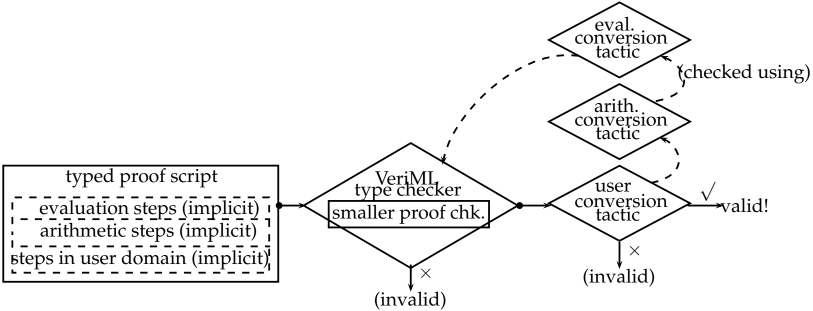
- separate included proof scripts out
- know earlier whether they're correct
- quite tedious for small (one-tactic) scripts

New idea: static proof scripts



- after transformation, static evaluation is possible
- type information makes transformation easy
- more-or-less transparent to the user

Resulting picture



Our toolbox

Higher-order logic

λHOL

$$
\begin{array}{rrcl}
\text{(logical terms)} & t & ::= & \textit{proof object constructors } \pi \\
 & & | & \textit{propositions } P \\
 & & | & \textit{natural numbers, functions etc.} \\
 & & | & \textit{sorts and types} \\
\text{(environments)} & \Phi & ::= & \bullet \mid \Phi,\ x : t \\
\text{(contextual terms)} & T & ::= & [\Phi]\, t
\end{array}
$$

VeriML

ML + dependent programming over contextual terms $[\Phi]\, t$ of λHOL

$$\begin{aligned} \tau \;::=\; & (\text{normal ML types}) \\ \mid\; & (X : T) \to \tau \\ \mid\; & \langle X : T \rangle \times \tau \\ \mid\; & (\phi : \mathsf{ctx}) \to \tau \end{aligned}$$

VeriML

ML + dependent programming over contextual terms $[\Phi]\,t$ of λHOL

$$
\begin{aligned}
\tau \;::=\;& \text{(normal ML types)} \\
\mid\;& (X : T) \to \tau \\
\mid\;& \langle X : T \rangle \times \tau \\
\mid\;& (\phi : \mathsf{ctx}) \to \tau
\end{aligned}
$$

dependent function over logical term

VeriML

ML + dependent programming over contextual terms $[\Phi]\,t$ of λHOL

$$\begin{array}{rcl} \tau & ::= & \text{(normal ML types)} \\ & | & (X : T) \to \tau \\ & | & \langle X : T \rangle \times \tau \\ & | & (\phi : \mathsf{ctx}) \to \tau \end{array}$$

dependent product of logical term

VeriML

ML + dependent programming over contextual terms $[\Phi]\, t$ of λHOL

$$\begin{aligned} \tau \ ::= & \ \text{(normal ML types)} \\ & \mid (X : T) \to \tau \\ & \mid \langle X : T \rangle \times \tau \\ & \mid (\phi : \mathsf{ctx}) \to \tau \end{aligned}$$

polymorphism over logic environments

VeriML

ML + dependent programming over contextual terms $[\Phi]\, t$ of λHOL

$$\begin{array}{rcl} \tau & ::= & \text{(normal ML types)} \\ & | & (X : T) \to \tau \\ & | & \langle X : T \rangle \times \tau \\ & | & (\phi : \mathsf{ctx}) \to \tau \end{array}$$

VeriML: Typed Computation of Logical Terms inside a Language with Effects, Antonis Stampoulis and Zhong Shao, ICFP 2010

Pattern matching over terms

$$
\begin{array}{l}
\textsf{match } P \textsf{ with} \\
\quad \begin{array}{lll}
 & P_1 \to P_2 & \mapsto \quad \cdots \\
\mid & P_1 \land P_2 & \mapsto \quad \cdots
\end{array}
\end{array}
$$

Pattern matching over contexts

Look into logical environment to extract hypotheses, etc.

$$
\begin{array}{l}
\mathsf{assumption} : (\phi : \mathsf{ctx}, P : \mathsf{Prop}) \to \mathsf{option}\ \langle P \rangle \\
\mathsf{assumption}\ \phi\ P = \\
\quad \mathsf{ctxcase}\ \phi\ \mathsf{of} \\
\qquad \begin{array}{lll}
 & \phi',\ H : P & \mapsto\ \mathsf{return}\ \langle H \rangle \\
\mid & \phi',\ H : (P' : \mathsf{Prop}) & \mapsto\ \mathsf{assumption}\ \phi'\ P \\
\mid & \phi',\ x : (T : \mathsf{Type}) & \mapsto\ \mathsf{assumption}\ \phi'\ P
\end{array}
\end{array}
$$

Proof-erasure semantics

- Disallow pattern matching on proof objects
- Therefore proof objects can't influence evaluation
- Evaluation under proof erasure still guarantees that valid proof objects exist
- Trivial based on type safety

The conversion rule

Conversion rule

$$\frac{\Phi \vdash \pi : P \qquad P =_R P'}{\Phi \vdash \pi : P'}\ \textsc{conv}$$

- permeates metatheory of the logic
- "hardcoded" tactic to check $P =_R P'$ implicitly

Throwing conversion away: explicit equality

CONV

$$\frac{\Phi \vdash \pi : P \qquad P =_R P'}{\Phi \vdash \pi : P'}$$

$$\downarrow$$

$$\frac{\Phi \vdash t_1 : \mathcal{K} \qquad \begin{array}{c}\Phi,\ x : \mathcal{K} \vdash P : \mathsf{Prop} \\ \Phi \vdash \pi : P[t_1/x]\end{array} \qquad \Phi \vdash \pi' : t_1 = t_2}{\Phi \vdash \mathsf{leibniz}\ (x : \mathcal{K}.P)\ \pi\ \pi' : P[t_2/x]}$$

$$\frac{\Phi,\ x : \mathcal{K} \vdash d : \mathcal{K}' \qquad \Phi \vdash d' : \mathcal{K}}{\Phi \vdash \mathsf{betaEq}\ (\lambda x : \mathcal{K}.d)\ d' : (\lambda x : \mathcal{K}.d)\ d' = d[d'/x]}$$

Getting conversion back

Write a tactic that decides whether $P = P'$ and returns a proof object if yes; evaluate it under proof-erasure semantics.

Getting conversion back

Write a tactic that decides whether $P = P'$ and returns a proof object if yes; evaluate it under proof-erasure semantics.

Uses three functions:

$$\begin{array}{lcl} \mathsf{whnf} & : & (\phi : \mathsf{ctx}, T : \mathsf{Type}, t : T) \to \langle t' : T, \mathsf{pf} : t = t' \rangle \\ \mathsf{equal?} & : & (\phi : \mathsf{ctx}, T : \mathsf{Type}, t_1 : T, t_2 : T) \to \mathsf{option}\ \langle t_1 = t_2 \rangle \\ \mathsf{conversion} & : & (\phi : \mathsf{ctx}, P : \mathsf{Prop}, P' : \mathsf{Prop}, \mathsf{pf} : P, \mathsf{pf}' : P = P') \to \\ & & \langle \mathsf{pf} : P' \rangle \end{array}$$

Getting conversion back

Write a tactic that decides whether $P = P'$ and returns a proof object if yes; evaluate it under proof-erasure semantics.

Uses three functions:

whnf	:	$(\phi : \mathsf{ctx}, T : \mathsf{Type}, t : T) \rightarrow \langle t' : T, \mathsf{pf} : t = t' \rangle$
equal?	:	$(\phi : \mathsf{ctx}, T : \mathsf{Type}, t_1 : T, t_2 : T) \rightarrow \mathsf{option}\ \langle t_1 = t_2 \rangle$
conversion	:	$(\phi : \mathsf{ctx}, P : \mathsf{Prop}, P' : \mathsf{Prop}, \mathsf{pf} : P, \mathsf{pf}' : P = P') \rightarrow \langle \mathsf{pf} : P' \rangle$

Simplify to weak-head normal form.

Getting conversion back

Write a tactic that decides whether $P = P'$ and returns a proof object if yes; evaluate it under proof-erasure semantics.

Uses three functions:

whnf	:	$(\phi : \mathsf{ctx}, T : \mathsf{Type}, t : T) \to \langle t' : T, \mathsf{pf} : t = t' \rangle$
equal?	:	$(\phi : \mathsf{ctx}, T : \mathsf{Type}, t_1 : T, t_2 : T) \to \mathsf{option}\ \langle t_1 = t_2 \rangle$
conversion	:	$(\phi : \mathsf{ctx}, P : \mathsf{Prop}, P' : \mathsf{Prop}, \mathsf{pf} : P, \mathsf{pf}' : P = P') \to \langle \mathsf{pf} : P' \rangle$

Traverse both terms and check equality; always simplify through whnf

Getting conversion back

Write a tactic that decides whether $P = P'$ and returns a proof object if yes; evaluate it under proof-erasure semantics.

Uses three functions:

whnf	:	$(\phi : \mathsf{ctx}, T : \mathsf{Type}, t : T) \rightarrow \langle t' : T, \mathsf{pf} : t = t' \rangle$
equal?	:	$(\phi : \mathsf{ctx}, T : \mathsf{Type}, t_1 : T, t_2 : T) \rightarrow \mathsf{option}\ \langle t_1 = t_2 \rangle$
conversion	:	$(\phi : \mathsf{ctx}, P : \mathsf{Prop}, P' : \mathsf{Prop}, \mathsf{pf} : P, \mathsf{pf}' : P = P') \rightarrow \langle \mathsf{pf} : P' \rangle$

Convert a proof object to a proof of an equivalent proposition. Uses equal? to do proof of $P = P'$.

Weak-head normal form

$$
\begin{array}{l}
\mathsf{whnf} : (\phi : \mathsf{ctx}, T : \mathsf{Type}, t : T) \to \langle t' : T, \mathsf{pf} : t = t' \rangle \\
\mathsf{whnf}\ \phi\ T\ t = \\
\quad \mathsf{holcase}\ t\ \mathsf{of} \\
\quad (t_1 : T' \to T)\ (t_2 : T') \mapsto \\
\quad\quad \mathsf{let}\ \langle t_1',\ \mathsf{pf}_1 \rangle = \mathsf{whnf}\ \phi\ (T' \to T)\ t_1\ \mathsf{in} \\
\quad\quad \mathsf{holcase}\ t_1'\ \mathsf{of} \\
\quad\quad\quad\quad \begin{array}{lcl} \lambda x : T'.t_f & \mapsto & \langle [\phi]\, t_f / [x := t_2], \cdots \rangle \\ \mid t_1' & \mapsto & \langle [\phi]\, t_1'\ t_2, \cdots \rangle \end{array} \\
\quad \mid t \mapsto \langle t,\ \cdots \rangle
\end{array}
$$

Testing equality

$$\text{equal?} : (\phi : \text{ctx}, T : \text{Type}, t_1 : T, t_2 : T) \to \text{option } \langle t_1 = t_2 \rangle$$

$$
\begin{array}{l}
\text{equal? } \phi\ T\ t_1\ t_2\ = \\
\quad \text{holcase whnf } \phi\ T\ t_1,\ \text{whnf } \phi\ T\ t_2 \text{ of} \\
\qquad (t_a\ t_b), (t_c\ t_d) \mapsto \\
\qquad\quad \text{do } \langle \text{pf}_1 \rangle \leftarrow \text{equal? } t_a\ t_c \\
\qquad\qquad \langle \text{pf}_2 \rangle \leftarrow \text{equal? } t_b\ t_d \\
\qquad\qquad \text{return } \langle \cdots\ \textit{proof of}\ \ t_a\ t_b = t_c\ t_d \cdots \rangle \\
\qquad \mid (\lambda x : T.t_1), (\lambda x : T.t_2) \mapsto \\
\qquad\quad \text{do } \langle \text{pf} \rangle \leftarrow \text{equal? } [\phi,\ x : T]\ t_1\ t_2 \\
\qquad\qquad \text{return } \langle \cdots\ \textit{proof of}\ \ \lambda x : T.t_1 = \lambda x : T.t_2 \cdots \rangle \\
\qquad \ldots
\end{array}
$$

Lifting proof objects to VeriML

From a proof object in the logic with conversion, get an equivalent typed proof script in VeriML.

Lifting proof objects to VeriML

constructor	to tactic	of type
$\lambda x : P.\pi$	$\mathsf{Assume}\ e$	$\langle [\phi,\ H : P]\, P' \rangle \to \langle P \to P' \rangle$
$\pi_1\ \pi_2$	$\mathsf{Apply}\ e_1\ e_2$	$\langle P \to P' \rangle \to \langle P \rangle \to \langle P' \rangle$
$\lambda x : \mathcal{K}.\pi$	$\mathsf{Intro}\ e$	$\langle [\phi,\ x : T]\, P' \rangle \to \langle \forall x : T, P' \rangle$
$\pi\ d$	$\mathsf{Inst}\ e\ a$	$\langle \forall x : T, P \rangle \to (a : T) \to \langle P/[x := a] \rangle$
c	$\mathsf{Lift}\ c$	$(H : P) \to \langle P \rangle$
(conversion)	$\mathsf{Conversion}$	$\langle P \rangle \to \langle P = P' \rangle \to \langle P' \rangle$

Refinements:

- use conversion implicitly
- use type inferrence
- call equal? statically

What did we gain?
Compared to proof objects

- conversion not part of proof checker
- simpler logic
- convertibility can be extended by user, safely
- proof consumer decides tradeoff of trust versus resources (proof erasure semantics or not)
- essentially: proof consumer adjusts conversion rule at will!

What did we gain?
Compared to proof scripts

- increased static checking
- can be further extended by user
- example: get proof “skeleton” to work first, do expensive proof search last

Stacking conversions

Stacking conversions

Idea: use simpler conversion tactics to implicitly prove all obligations in more complicated ones!

Current stack

basic support
naive equality
union-find equality
naive arithmetic
better arithmetic

Current stack

basic support
naive equality
union-find equality
naive arithmetic
better arithmetic

syntactic equality as previously shown, parametric over `whnf`-like simplifier

Current stack

basic support
naive equality
union-find equality
naive arithmetic
better arithmetic

- isolate hypotheses like $x = y$ from context
- blindly rewrite x into y
- bad strategy, but...

Current stack

basic support
naive equality
union-find equality
naive arithmetic
better arithmetic

- standard textbook implementation of equality with uninterpreted functions
- uses imperative union-find data structures
- all proofs handled by naive equality

Current stack

basic support
naive equality
union-find equality
naive arithmetic
better arithmetic

- use existing conversion to simplify proofs of properties
- naive rewriting based on commutativity and distributivity

Current stack

basic support
naive equality
union-find equality
naive arithmetic
better arithmetic

- more sophisticated arithmetic simplifications
- canonical form of polynomials
- use naive arithmetic to simplify proofs

Summary

Summary: extensible conversion rule

- A way to extend proof checker for proof objects
- and static checking for proof scripts
- ... through user-defined code
- ... written in a general programming model
- ... without risking soundness
- ... with no metatheory additions to the logic
- ... actually, with reductions
- Using a language for type-safe tactics: VeriML
- Extensive metatheory and prototype implementation

Future work

- compile VeriML to ML
- use hash-consing in conversion
- term nets to know when specific conversion applies
- extend to full CIC

Thanks!

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

(talk to me for draft or implementation!)