VeriML: Typed Computation of Logical Terms inside a Language with Effects

Antonis Stampoulis Zhong Shao

Department of Computer Science, Yale University

ICFP 2010

Proof assistants are becoming popular in the PL community!

Proof assistants are becoming popular in the PL community!

- Anything from metatheory proofs

Proof assistants are becoming popular in the PL community!

- Anything from metatheory proofs
- ... to verified compilers (CompCert by Leroy et al.)

Proof assistants are becoming popular in the PL community!

- Anything from metatheory proofs
- ... to verified compilers (CompCert by Leroy et al.)
- ... and verified operating systems (seL4 verification by Klein et al.)

How to go to the next step?

How to go to the next step?

- Manual proof effort needs to be reduced

How to go to the next step?

- Manual proof effort needs to be reduced
- CompCert: proofs are 44% of the development (executable code 1045 lines, proofs 16543)

How to go to the next step?

- Manual proof effort needs to be reduced
- CompCert: proofs are 44% of the development (executable code 1045 lines, proofs 16543)
- proof to executable code ratio is about 16 to 1
- seL4: about 11 to 1

Maybe inherrent complexity of proofs?

Maybe inherrent complexity of proofs?

- Not necessarily
- e.g. Chlipala (POPL 2010): verified compiler where only 25% of development is proofs

What's the trick?

Focus less on writing proof scripts,
focus more on writing tactics.

Proof scripts?

A series of applications of tactics.

Proof scripts?

A series of applications of tactics.

Tactics?

- No clear definition
- Very informally: functions that generate (part of a) proof for specific kinds of goals
- Reality much more complicated (Asperti et al. A New Type for Tactics)

More liberal definition:

tactics are functions that
operate on propositions and proofs

and produce other proofs

More liberal definition:

tactics are functions that
operate on propositions and proofs
(in general: on logical terms)

and produce other proofs

More liberal definition:

tactics are functions that
operate on propositions and proofs

(in general: on logical terms)

(and potentially on other things as well)

and produce other proofs

(and potentially other things as well)

So the motto?

instead of scripts with lots of general-purpose tactics
develop domain-specific tactics thus smaller scripts

So the motto?

instead of scripts with lots of general-purpose tactics
develop domain-specific tactics thus smaller scripts

- more reusable than proof scripts
- more modular – thus more scalable
- e.g. to prove Hoare triples $\{P\}\ c\ \{Q\}$:
  - tactic to decide arithmetic formulas
  - tactic to do VC gen
  - compose one with the other for Hoare triples tactic

Why not more popular?

Claim: Language support for writing tactics relatively poor!

Why not more popular?

Claim: Language support for writing tactics relatively poor!

- no good way to specify what tactics do:
  - arguments? goals they operate on? etc.
  - rely on documentation
  - hurts composability of tactics!
- OR trade expressivity for being able to specify them

Need language to specify and implement tactics!

Sounds familiar...

Sounds familiar...

Robin Milner, circa 1973 :
ML
(originally to write tactics for LCF)

But programming language theory has evolved!

But programming language theory has evolved!

Leverage dependent types – as a step towards more detailed specifications

Our contribution: VeriML

- ML core calculus (keep expressivity)
- extended with dependent types for logical terms
- but can still "operate on" logical terms
- use a logic similar to CIC (no dependent types!) with explicit proof objects
- type system that guarantees validity of logical terms and safe handling of binding
- proof of type safety
- prototype implementation

An example: equality tactic

Based on a list of equations like

$$x = y, y = z, w = q, w = z$$

decide whether e.g. $x = q$

Type for this tactic?

Based on a list of equations like
$x = y, y = z, w = q, w = z$
decide whether e.g. $x = q$

$$\text{equality} \quad : \quad \text{list}\,(\,\text{term} \;*\; \text{term} \;*\; \text{proof}\,) \rightarrow$$
$$\text{term} \;\rightarrow\; \text{term} \;\rightarrow\; \text{option}\;\; \text{proof}$$

Type for this tactic?

Based on a list of equations like
$x = y, y = z, w = q, w = z$
decide whether e.g. $x = q$

$$\begin{aligned} \text{equality} \quad : \quad & \text{list}\,(\,\text{term} \ * \ \text{term} \ * \ \text{proof}\,) \longrightarrow \\ & \text{term} \ \longrightarrow \ \text{term} \ \longrightarrow \ \text{option} \ \ \text{proof} \end{aligned}$$

But

Type for this tactic?

Based on a list of equations like
$x = y, y = z, w = q, w = z$
decide whether e.g. $x = q$

$$\text{equality} \quad : \quad \text{list}\ (\ T : Set \ * \ a : T \ * \ b : T \ * \ \text{proof}\ ) \rightarrow$$
$$T : Set \ \rightarrow \ x : T \ \rightarrow \ y : T \ \rightarrow \ \text{option} \ \ \text{proof}$$

But

- terms should be of the same type T ($=$ Nat, List, ...)

Type for this tactic?

Based on a list of equations like
$x = y, y = z, w = q, w = z$
decide whether e.g. $x = q$

equality : list ($T : Set \; * \; a : T \; * \; b : T \; * \; pf : a = b$) $\rightarrow$
$T : Set \; \rightarrow \; x : T \; \rightarrow \; y : T \; \rightarrow$ option ($pf' : x = y$)

But

- terms should be of the same type T (= Nat, List, ...)
- the proof should prove that they're equal

$$\text{equality} \quad : \quad \text{list}\ (\ T : Set\ *\ a : T\ *\ b : T\ *\ pf : a = b\ ) \rightarrow$$
$$T : Set\ \rightarrow\ x : T\ \rightarrow\ y : T\ \rightarrow\ \text{option}\ (\ pf' : x = y\ )$$

Better specification means

- more composable (know input/outputs precisely)
- more errors can be caught at compile time

How to implement?

Union-find data structure

- each equivalence class has a representative
- each term has a parent term
- if parent term equal to term, it's the representative
- merge representatives on new equality

How to implement?

Union-find data structure

- each equivalence class has a representative
- each term has a parent term
- if parent term equal to term, it's the representative
- merge representatives on new equality

Assume:

$$
\begin{array}{lll}
\text{uftype} & & \textit{type for union-find data structure} \\
\text{ufGet} & : & \text{uftype} \longrightarrow (\, base : T \,) \longrightarrow \text{option} \,(\, parent : T \,* \\
& & \qquad\qquad\qquad\qquad\qquad pf : base = parent \,) \\
\text{ufSet} & : & \text{uftype} \longrightarrow (\, base : T \,*\, parent : T \,* \\
& & \qquad\qquad pf : base = parent \,) \longrightarrow \text{unit}
\end{array}
$$

Implementation of find

Find the representative of the equiv. class of a term

```
find : uftype → ( base : T ) → ( rep : T )
find uf base =
    match ufGet uf base with
          None ↦
             ufSet ( base )( base );
            ( base )
      |   Some ( parent ) ↦
             holcase parent with
                 base   ↦  ( base )
                | _     ↦   let rep  =
                                 find uf parent
                            in ( rep )
```

Implementation of find

Find the representative of the equiv. class of a term

$$
\begin{array}{l}
\mathsf{find} : \mathsf{uftype} \to (\, base : T\,) \to (\, rep : T * \underline{pf : base = rep}\,) \\
\mathsf{find}\ \mathsf{uf}\ base = \\
\quad \mathbf{match}\ \mathsf{ufGet}\ \mathsf{uf}\ base\ \mathbf{with} \\
\qquad \mathsf{None} \mapsto \\
\qquad\quad \mathsf{ufSet}\,(\, base\,)(\, base, \underline{reflexivity\ base}\,); \\
\qquad\quad (\, base, \underline{reflexivity\ base}\,) \\
\quad \mid\ \ \mathsf{Some}\,(\, parent, \underline{pf}\,) \mapsto \\
\qquad\quad \mathbf{holcase}\ parent\ \mathbf{with} \\
\qquad\qquad base \mapsto (\, base, \underline{reflexivity\ base}\,) \\
\qquad\qquad \mid \_\_ \mapsto \mathbf{let}\ rep, \underline{pf'} = \\
\qquad\qquad\qquad\qquad\qquad \mathsf{find}\ \mathsf{uf}\ parent \\
\qquad\qquad\qquad \mathbf{in}\ (\, rep, \underline{transitivity\ pf\ pf'}\,)
\end{array}
$$

Implementation of find

Find the representative of the equiv. class of a term

$$\text{find} \;:\; \text{uftype} \;\rightarrow (\, base :\; T\,) \rightarrow (\, rep :\; T * \underline{\;pf : base = rep\;}\,)$$

$$\text{find uf } base =$$

$\quad$ **match** ufGet uf $base$ **with**

$\qquad$ None $\mapsto$

$\qquad\quad$ ufSet $(\, base\,)(\, base\,,\, \underline{reflexivity\ base : base = base}\,)$;

$\qquad\quad$ $(\, base\,,\, \underline{reflexivity\ base : base = base}\,)$

$\quad$ | $\quad$ Some $(\, parent\,,\, \underline{pf\qquad\qquad\qquad}\,) \mapsto$

$\qquad\quad$ **holcase** $parent$ **with**

$\qquad\qquad$ $base \;\mapsto\; (\, base\,,\, \underline{reflexivity\ base\qquad\qquad}\,)$

$\qquad\qquad$ | __ $\;\mapsto\;$ **let** $rep\,,\, \underline{pf'\qquad\qquad\qquad} =$

$\qquad\qquad\qquad\qquad\qquad\qquad$ find uf $parent$

$\qquad\qquad\qquad\quad$ **in** $(\, rep\,,\, \underline{transitivity\ pf\ pf'\qquad\qquad}\,)$

$$\text{ufSet} \;:\; \text{uftype} \;\rightarrow (\, base :\; T * \; parent :\; T * \; pf : base = parent\,) \rightarrow \text{unit}$$

Implementation of find

Find the representative of the equiv. class of a term

$$
\begin{array}{l}
\text{find} \;:\; \text{uftype} \;\rightarrow (\, base \,:\, T\,) \rightarrow (\, rep \,:\, T \,*\, \underline{pf : base = rep}\,) \\
\text{find uf } base \;= \\
\quad \textbf{match } \text{ufGet uf } base \textbf{ with} \\
\qquad \text{None} \mapsto \\
\qquad\quad \text{ufSet}\,(\, base\,)(\, base\,,\, \underline{reflexivity\ base : base = base}\,); \\
\qquad\quad (\, base\,,\, \underline{reflexivity\ base : base = base}\,) \\
\quad\; | \quad \text{Some}\,(\, parent\,,\, \underline{pf :\ base = parent}\,) \mapsto \\
\qquad\quad \textbf{holcase } parent \textbf{ with} \\
\qquad\qquad base \;\mapsto\; (\, base\,,\, \underline{reflexivity\ base \qquad\qquad}\,) \\
\qquad\qquad |\ \_\_ \;\mapsto\; \textbf{let } rep\,,\, \underline{pf' \qquad\qquad} = \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{find uf } parent \\
\qquad\qquad\qquad\quad \textbf{in } (\, rep\,,\, \underline{transitivity\ pf\ pf' \qquad\qquad}\,)
\end{array}
$$

$$
\begin{array}{l}
\text{ufGet} \;:\; \text{uftype} \;\rightarrow (\, base \,:\, T\,) \rightarrow \text{option}\,(\, parent \,:\, T \,* \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad pf : base = parent\,)
\end{array}
$$

Implementation of find

Find the representative of the equiv. class of a term

$$
\begin{array}{l}
\textsf{find} : \textsf{uftype} \to (\, base : T \,) \to (\, rep : T * \underline{pf : base = rep} \,) \\
\textsf{find uf } base = \\
\quad \textbf{match } \textsf{ufGet uf } base \textbf{ with} \\
\qquad \textsf{None} \mapsto \\
\qquad\quad \textsf{ufSet}(\, base \,)(\, base , \underline{reflexivity\ base : base = base} \,); \\
\qquad\quad (\, base , \underline{reflexivity\ base : base = base} \,) \\
\quad \mid \quad \textsf{Some}(\, parent , \underline{pf : base = parent} \,) \mapsto \\
\qquad\quad \textbf{holcase } parent \textbf{ with} \\
\qquad\qquad base \mapsto (\, base , \underline{reflexivity\ base : base = base} \,) \\
\qquad\qquad \mid \_ \mapsto \textbf{let } rep , \underline{pf'} = \\
\qquad\qquad\qquad\qquad \textsf{find uf } parent \\
\qquad\qquad\qquad \textbf{in } (\, rep , \underline{transitivity\ pf\ pf'} \,)
\end{array}
$$

Implementation of find

Find the representative of the equiv. class of a term

$$
\begin{array}{l}
\mathsf{find}\ :\ \mathsf{uftype}\ \to (\ base : T\ ) \to (\ rep : T * \underline{pf : base = rep}\ ) \\
\mathsf{find}\ \ \mathsf{uf}\ \ base\ = \\
\quad \textbf{match}\ \ \mathsf{ufGet}\ \ \mathsf{uf}\ \ base\ \ \textbf{with} \\
\qquad \mathsf{None}\ \mapsto \\
\qquad\quad \mathsf{ufSet}\ (\ base\ )(\ base\ ,\ \underline{reflexivity\ base : base = base}\ ); \\
\qquad\quad (\ base\ ,\ \underline{reflexivity\ base : base = base}\ ) \\
\quad\ |\quad \mathsf{Some}\ (\ parent\ ,\ \underline{pf : base = parent}\ ) \mapsto \\
\qquad\quad \textbf{holcase}\ \ parent\ \ \textbf{with} \\
\qquad\qquad base \quad \mapsto \quad (\ base\ ,\ \underline{reflexivity\ base : base = base}\ ) \\
\qquad\qquad |\ \_\_ \quad \mapsto \quad \textbf{let}\ \ rep\ \ ,\ \underline{pf' : parent = rep}\ = \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{find}\ \ \mathsf{uf}\ \ parent \\
\qquad\qquad\qquad\qquad \textbf{in}\ (\ rep\ ,\ \underline{transitivity\ pf\ \ pf'}\ )
\end{array}
$$

Implementation of find

Find the representative of the equiv. class of a term

$$
\begin{array}{l}
\text{find} : \text{uftype} \rightarrow (\mathit{base} : T) \rightarrow (\mathit{rep} : T * \underline{\mathit{pf} : \mathit{base} = \mathit{rep}}) \\
\text{find uf } \mathit{base} = \\
\quad \textbf{match } \text{ufGet uf } \mathit{base} \textbf{ with} \\
\qquad \text{None} \mapsto \\
\qquad\quad \text{ufSet}(\mathit{base})(\mathit{base}, \underline{\mathit{reflexivity\ base} : \mathit{base} = \mathit{base}}); \\
\qquad\quad (\mathit{base}, \underline{\mathit{reflexivity\ base} : \mathit{base} = \mathit{base}}) \\
\quad | \quad \text{Some}(\mathit{parent}, \underline{\mathit{pf} : \mathit{base} = \mathit{parent}}) \mapsto \\
\qquad\quad \textbf{holcase } \mathit{parent} \textbf{ with} \\
\qquad\qquad \mathit{base} \mapsto (\mathit{base}, \underline{\mathit{reflexivity\ base} : \mathit{base} = \mathit{base}}) \\
\qquad\qquad | \ \_ \mapsto \textbf{let } \mathit{rep}, \underline{\mathit{pf}' : \mathit{parent} = \mathit{rep}} = \\
\qquad\qquad\qquad\qquad\qquad \text{find uf } \mathit{parent} \\
\qquad\qquad\qquad \textbf{in } (\mathit{rep}, \underline{\mathit{transitivity\ pf\ pf}' : \mathit{base} = \mathit{rep}})
\end{array}
$$

- type checker would not allow to switch arguments to transitivity!

Another example

simplify :
A tactic that simplifies propositions like $P \wedge \mathsf{True}$ to P ,
recursively.

Implementation

$$
\begin{array}{lllll}
\multicolumn{5}{l}{\text{simplify } : (\, P : Prop \,) \rightarrow (\, P' : Prop \; * \; pf : P \leftrightarrow P' \,)} \\
\multicolumn{5}{l}{\text{simplify } P \; = \; \text{holcase } P \text{ with}} \\
 & P_1 \wedge \text{True} & \mapsto & \multicolumn{2}{l}{\text{let } P_1' \,,\; pf' \; = \; \text{simplify } \; P_1 \; \text{ in}} \\
 & & & \multicolumn{2}{l}{(\, P_1' \,,\; \cdots \,)} \\
| & P_1 \vee P_2 & \mapsto & \multicolumn{2}{l}{\text{let } P_1' \,,\; pf_1 \; = \; \text{simplify } \; P_1 \; \text{ in}} \\
 & & & \multicolumn{2}{l}{\text{let } P_2' \,,\; pf_2 \; = \; \text{simplify } \; P_2 \; \text{ in}} \\
 & & & \multicolumn{2}{l}{(\, P_1' \vee P_2' \,,\; \cdots \,)} \\
| & \forall x : Nat . P_1 & \mapsto & \multicolumn{2}{l}{\text{let } P_1' \,,\; pf' \; = \; \text{simplify } \; P_1 \; \text{ in}} \\
 & & & \multicolumn{2}{l}{(\, P_1' \,,\; \cdots \,)} \\
| & \_\_ & \mapsto & \multicolumn{2}{l}{(\, P \,,\; \cdots \,)}
\end{array}
$$

Implementation

$$
\begin{array}{l}
\mathsf{simplify}\ : (\ P : Prop\ ) \to (\ P' : Prop\ *\ pf : P \leftrightarrow P'\ ) \\
\mathsf{simplify}\ \ P\ =\ \mathsf{holcase}\ \ P\ \ \mathsf{with} \\
\begin{array}{llcl}
 & P_1 \wedge \mathsf{True} & \mapsto & \mathsf{let}\ \ P'_1\ ,\ pf'\ =\ \mathsf{simplify}\ \ P_1\ \ \mathsf{in} \\
 & & & (\ P'_1\ ,\ \cdots\ ) \\
| & P_1 \vee P_2 & \mapsto & \mathsf{let}\ \ P'_1\ ,\ pf_1\ =\ \mathsf{simplify}\ \ P_1\ \ \mathsf{in} \\
 & & & \mathsf{let}\ \ P'_2\ ,\ pf_2\ =\ \mathsf{simplify}\ \ P_2\ \ \mathsf{in} \\
 & & & (\ P'_1 \vee P'_2\ ,\ \cdots\ ) \\
| & \forall x : Nat.P_1 & \mapsto & \mathsf{let}\ \ P'_1\ ,\ pf'\ =\ \mathsf{simplify}\ \ P_1\ \ \mathsf{in} \\
 & & & (\ P'_1\ ,\ \cdots\ ) \\
| & \_\_ & \mapsto & (\ P\ ,\ \cdots\ )
\end{array}
\end{array}
$$

- oops: what if we could apply it to $\forall x : Nat.x = 3$
- variable x escapes into ill-formed $x = 3$

Solution

Type system should keep track of free variables environment of logical terms!

Provide substitution for free variables a term depends on, in the current environment

Solution

$$\text{simplify} \;:\; \underline{(\ \Phi : \text{context}\ )} \to (\ P :\ \underline{[\Phi]Prop}\ ) \to$$
$$(\ P' :\ \underline{[\Phi]Prop}\ *\ pf :\ \underline{[\Phi](P \leftrightarrow P')}\ )$$

$$\text{simplify}\ \underline{\ \Phi\ }\ P \;=\; \text{holcase}\ \ P\ \ \text{with}$$

$$\begin{array}{lll}
\quad P_1 \wedge \text{True} & \mapsto & \text{let}\ \ P_1'\ ,\ pf' \;=\; \text{simplify}\ \underline{\ \Phi\ }\ P_1\ \ \text{in} \\
 & & (\ P_1'\ ,\ \cdots\ ) \\
\mid\quad P_1 \vee P_2 & \mapsto & \text{let}\ \ P_1'\ ,\ pf_1 \;=\; \text{simplify}\ \underline{\ \Phi\ }\ P_1\ \ \text{in} \\
 & & \text{let}\ \ P_2'\ ,\ pf_2 \;=\; \text{simplify}\ \underline{\ \Phi\ }\ P_2\ \ \text{in} \\
 & & (\ P_1' \vee P_2'\ ,\ \cdots\ ) \\
\mid\quad \forall x : Nat.P_1 & \mapsto & \\
\end{array}$$

$$\quad \text{let}\ \ P_1'\ ,\ pf' \;=\; \text{simplify}\ \underline{(\Phi, x : Nat)}\ P_1\ \ \text{in}$$
$$\quad (\ P_1'\ ,\ \cdots\ )$$

$$\begin{array}{lll}
\mid\quad __ & \mapsto & (\ P\ ,\ \cdots\ )
\end{array}$$

Solution

$$
\begin{array}{l}
\text{simplify} \;:\; \underline{(\;\Phi : \text{context}\;)} \rightarrow (\;P : \underline{[\Phi]Prop}\;) \rightarrow \\
\qquad\qquad\qquad\qquad (\;P' : \underline{[\Phi]Prop} \;*\; pf : \underline{[\Phi](P \leftrightarrow P')}\;) \\
\text{simplify}\;\underline{\;\Phi\;}\;P \;=\; \text{holcase}\;\;P\;\;\text{with} \\
\qquad P_1 \wedge \text{True} \quad\mapsto\quad \text{let}\;\;P_1'\;,\;pf' \;=\; \text{simplify}\;\underline{\;\Phi\;}\;P_1\;\;\text{in} \\
\qquad\qquad\qquad\qquad\quad (\;P_1'\;,\;\cdots\;) \\
\;\mid\quad P_1 \vee P_2 \quad\mapsto\quad \text{let}\;\;P_1'\;,\;pf_1 \;=\; \text{simplify}\;\underline{\;\Phi\;}\;P_1\;\;\text{in} \\
\qquad\qquad\qquad\qquad\quad \text{let}\;\;P_2'\;,\;pf_2 \;=\; \text{simplify}\;\underline{\;\Phi\;}\;P_2\;\;\text{in} \\
\qquad\qquad\qquad\qquad\quad (\;P_1' \vee P_2'\;,\;\cdots\;) \\
\;\mid\quad \forall x : Nat.P_1 \quad\mapsto \\
\qquad\quad \text{let}\;\;P_1'\;,\;pf' \;=\; \text{simplify}\;\underline{(\Phi, x : Nat)}\;P_1\;\;\text{in} \\
\qquad\quad (\;P_1'\;,\;\cdots\;) \\
\;\mid\quad \_ \qquad\qquad\quad \mapsto\;\;(\;P\;,\;\cdots\;)
\end{array}
$$

$P_1' : [\Phi, x : Nat]Prop$ needs a substitution into $[\Phi]Prop$

Behind the scenes

$$\tau ::= \mathsf{int} \mid \mathsf{unit} \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 + \tau_2 \mid \mu\alpha.\tau \mid \forall\alpha.\tau \mid \mathsf{ref}\ \tau \mid \cdots$$

Behind the scenes

$$
\begin{aligned}
\tau ::= {} & \mathsf{int} \mid \mathsf{unit} \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid \mu\alpha.\tau \mid \forall\alpha.\tau \mid \mathsf{ref}\ \tau \mid \cdots \\
& \mid \Pi\ \mathit{\Phi} :\ \mathit{context}\ .\tau
\end{aligned}
$$

Behind the scenes

$$
\begin{aligned}
\tau ::= {} & \mathsf{int} \mid \mathsf{unit} \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid \mu\alpha.\tau \mid \forall\alpha.\tau \mid \mathsf{ref}\ \tau \mid \cdots \\
& \mid \Pi\ \Phi : \ context\ .\tau \\
& \mid \Pi\ X : \ [\Phi]T\ .\tau
\end{aligned}
$$

Behind the scenes

$$
\begin{aligned}
\tau ::= {} & \mathsf{int} \mid \mathsf{unit} \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid \mu\alpha.\tau \mid \forall\alpha.\tau \mid \mathsf{ref}\ \tau \mid \cdots \\
& \mid \Pi\ \Phi :\ \mathit{context}\ .\tau \\
& \mid \Pi\ X :\ [\Phi]T\ .\tau \\
& \mid \Sigma\ X :\ [\Phi]T\ .\tau
\end{aligned}
$$

Behind the scenes

$$\begin{aligned}\tau ::= {} & \mathsf{int} \mid \mathsf{unit} \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 + \tau_2 \mid \mu\alpha.\tau \mid \forall\alpha.\tau \mid \mathsf{ref}\ \tau \mid \cdots \\ & \mid \Pi\ \mathit{\Phi} : \ \mathit{context}\ .\tau \\ & \mid \Pi\ X : \ [\mathit{\Phi}]T\ .\tau \\ & \mid \Sigma\ X : \ [\mathit{\Phi}]T\ .\tau\end{aligned}$$

$$e ::= \cdots$$

Behind the scenes

$$
\begin{aligned}
\tau ::= {} & \mathsf{int} \mid \mathsf{unit} \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid \mu\alpha.\tau \mid \forall\alpha.\tau \mid \mathsf{ref}\ \tau \mid \cdots \\
& \mid \Pi\ \Phi : \ \mathit{context}\ .\tau \\
& \mid \Pi\ X : \ [\Phi]T\ .\tau \\
& \mid \Sigma\ X : \ [\Phi]T\ .\tau
\end{aligned}
$$

$$
\begin{aligned}
e ::= {} & \cdots \\
& \mid \lambda\ \Phi : \ \mathit{context}\ .e \ \mid e\ \ \Phi
\end{aligned}
$$

Behind the scenes

$$
\begin{aligned}
\tau ::= {} & \mathsf{int} \mid \mathsf{unit} \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 + \tau_2 \mid \mu\alpha.\tau \mid \forall\alpha.\tau \mid \mathsf{ref}\ \tau \mid \cdots \\
& \mid \Pi\ \mathit{\Phi} :\ \mathit{context}\ .\tau \\
& \mid \Pi\ X :\ [\mathit{\Phi}]T\ .\tau \\
& \mid \Sigma\ X :\ [\mathit{\Phi}]T\ .\tau
\end{aligned}
$$

$$
\begin{aligned}
e ::= {} & \cdots \\
& \mid \lambda\ \mathit{\Phi} :\ \mathit{context}\ .e \ \mid e\ \ \mathit{\Phi} \\
& \mid \lambda\ X :\ [\mathit{\Phi}]T\ .e \ \mid e\ \ [\mathit{\Phi}]T
\end{aligned}
$$

Behind the scenes

$$\tau ::= \mathsf{int} \mid \mathsf{unit} \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 + \tau_2 \mid \mu\alpha.\tau \mid \forall\alpha.\tau \mid \mathsf{ref}\ \tau \mid \cdots$$
$$\mid \Pi\ \Phi :\ \mathit{context}\ .\tau$$
$$\mid \Pi\ X :\ [\Phi]T\ .\tau$$
$$\mid \Sigma\ X :\ [\Phi]T\ .\tau$$

$$e ::= \cdots$$
$$\mid \lambda\ \Phi :\ \mathit{context}\ .e\ \mid e\ \ \Phi$$
$$\mid \lambda\ X :\ [\Phi]T\ .e\ \mid e\ \ [\Phi]T$$
$$\mid \langle\ [\Phi]T\ ,\ e\rangle \mid\ \mathsf{let}\ \langle\ X\ ,\ \mathsf{y}\ \rangle = e\ \mathsf{in}\ e'$$

Behind the scenes

$$\tau ::= \mathsf{int} \mid \mathsf{unit} \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid \mu\alpha.\tau \mid \forall\alpha.\tau \mid \mathsf{ref}\ \tau \mid \cdots$$
$$\mid \Pi\ \Phi : \ context\ .\tau$$
$$\mid \Pi\ X : \ [\Phi]T\ .\tau$$
$$\mid \Sigma\ X : \ [\Phi]T\ .\tau$$

$$e ::= \cdots$$
$$\mid \lambda\ \Phi : \ context\ .e \ \mid e \ \ \Phi$$
$$\mid \lambda\ X : \ [\Phi]T\ .e \ \mid e \ \ [\Phi]T$$
$$\mid \langle\ [\Phi]T\ ,\ e\rangle \mid \ \mathsf{let}\ \langle\ X\ ,\ \mathsf{y}\ \rangle = e\ \mathsf{in}\ e'$$
$$\mid \ \mathsf{holcase}\ \ [\Phi]T\ \ \mathsf{with}\ (\ T_1\ \mapsto e_1\ ) \cdots (\ T_n\ \mapsto e_n\ )$$

Behind the scenes

$$\tau ::= \mathsf{int} \mid \mathsf{unit} \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid \mu\alpha.\tau \mid \forall\alpha.\tau \mid \mathsf{ref}\ \tau \mid \cdots$$
$$\mid \Pi\ \Phi : \ \mathit{context}\ .\tau$$
$$\mid \Pi\ X : \ [\Phi]T\ .\tau$$
$$\mid \Sigma\ X : \ [\Phi]T\ .\tau$$

$$e ::= \cdots$$
$$\mid \lambda\ \Phi : \ \mathit{context}\ .e \ \mid e\ \ \Phi$$
$$\mid \lambda\ X : \ [\Phi]T\ .e \ \mid e\ \ [\Phi]T$$
$$\mid \langle\ [\Phi]T\ ,\ e\rangle \mid\ \mathsf{let}\ \langle\ X\ ,\ \mathsf{y}\ \rangle = e\ \mathsf{in}\ e'$$
$$\mid\ \mathsf{holcase}\ \ [\Phi]T\ \ \mathsf{with}\ (\ T_1\ \mapsto e_1\ ) \cdots (\ T_n\ \mapsto e_n\ )$$

Full details of type system and metatheory in the paper and TR!

Implementation

- prototype in OCaml
- about 5k lines, trusted base is 800 lines
- examples:
  - first-order tautologies prover
  - conversion to NNF
  - equality with uninterpreted functions
- download from
  `http://flint.cs.yale.edu/publications/veriml.html`

Comparison with Coq

Three ways to write tactics:

- ML
- LTac
- proof-by-reflection

Comparison with Coq

Three ways to write tactics:

- ML (untyped tactics, high barrier: requires knowledge of implementation internals)
- LTac
- proof-by-reflection

Comparison with Coq

Three ways to write tactics:

- ML (untyped tactics, high barrier: requires knowledge of implementation internals)
- LTac (untyped tactics, somewhat limited programming model)
- proof-by-reflection

Comparison with Coq

Three ways to write tactics:

- ML (untyped tactics, high barrier: requires knowledge of implementation internals)
- LTac (untyped tactics, somewhat limited programming model)
- proof-by-reflection (strong static guarantees but very limited programming model)

Comparison with Coq

Three ways to write tactics:

- ML (untyped tactics, high barrier: requires knowledge of implementation internals)
- LTac (untyped tactics, somewhat limited programming model)
- proof-by-reflection (strong static guarantees but very limited programming model)

VeriML enables all points between no static guarantees to strong ones, yet with full ML programming model

Conclusion

- new language design with first-class support for rich logical framework
- enables more modular development of tactics
- type safety guarantees valid terms are generated

Future work

- type reconstruction, implicit parameters
- interactive proof support
- SAT-solving

Thank you!

Backup slides

Implementation of union

$$
\begin{array}{l}
\textsf{union} \;:\; \textsf{uftype} \;\rightarrow (\, a : T \,*\, b : T\,) \rightarrow \textsf{unit} \\
\textsf{union} \;\; \textsf{uf} \,(\, a \,,\; b \,) = \\
\quad \textsf{let} \;\; repA \;=\; \textsf{find} \;\; \textsf{uf} \;\; a \;\; \textsf{in} \\
\quad \textsf{let} \;\; repB \;=\; \textsf{find} \;\; \textsf{uf} \;\; B \;\; \textsf{in} \\
\quad \textsf{holcase} \;\; repA \;\; \textsf{with} \\
\qquad repB \quad \mapsto \quad () \\
\qquad \mid \_\_ \qquad \mapsto \quad \textsf{ufSet} \;\; repA \,(\, repB \,)
\end{array}
$$

Implementation of union

$$
\begin{array}{l}
\textsf{union} \; : \; \textsf{uftype} \rightarrow (\, a : T * b : T * pf : a = b \,) \rightarrow \textsf{unit} \\
\textsf{union} \;\; \textsf{uf} \, (\, a \,, \, b \,, \, pf \,) = \\
\quad \textsf{let} \;\; repA \,, \; pfA \qquad\qquad = \;\textsf{find} \;\; \textsf{uf} \;\; a \;\; \textsf{in} \\
\quad \textsf{let} \;\; repB \,, \; pfB \qquad\qquad = \;\textsf{find} \;\; \textsf{uf} \;\; B \;\; \textsf{in} \\
\quad \textsf{holcase} \;\; repA \;\; \textsf{with} \\
\qquad repB \quad \mapsto \quad () \\
\qquad \mid \_\_ \qquad \mapsto \quad \textsf{ufSet} \;\; repA \, (\, repB \,, \; \cdots \qquad\qquad)
\end{array}
$$

Implementation of union

$$
\begin{array}{l}
\text{union} \ : \ \text{uftype} \ \rightarrow (\ a : T \ * \ b : T \ * \ pf : a = b\ ) \rightarrow \ \text{unit} \\
\text{union} \ \ \text{uf}\ (\ a\ ,\ b\ ,\ pf\ ) = \\
\quad \text{let} \ \ repA\ ,\ pfA \ : a = repA \ = \ \text{find} \ \ \text{uf} \ \ a \ \ \text{in} \\
\quad \text{let} \ \ repB\ ,\ pfB \ : b = repB \ = \ \text{find} \ \ \text{uf} \ \ B \ \ \text{in} \\
\quad \text{holcase} \ \ repA \ \ \text{with} \\
\qquad repB \quad \mapsto \quad () \\
\qquad |\ \_\_ \qquad \mapsto \quad \text{ufSet} \ \ repA\ (\ repB\ ,\ \cdots \qquad\qquad)
\end{array}
$$

Implementation of union

$$
\begin{array}{l}
\text{union} \;:\; \text{uftype} \;\longrightarrow (\; a : T \,*\, b : T \,*\, pf : a = b \;) \longrightarrow \;\text{unit} \\
\text{union}\;\; \text{uf} \,(\; a \,,\; b \,,\; pf \,) = \\
\quad \text{let}\;\; repA \,,\; pfA \;: a = repA \;\; = \;\; \text{find}\;\; \text{uf}\;\; a \;\; \text{in} \\
\quad \text{let}\;\; repB \,,\; pfB \;: b = repB \;\; = \;\; \text{find}\;\; \text{uf}\;\; B \;\; \text{in} \\
\quad \text{holcase}\;\; repA \;\; \text{with} \\
\qquad repB \quad \mapsto \quad () \\
\qquad |\; \_\_ \qquad \mapsto \quad \text{ufSet}\;\; repA \,(\; repB \,,\; \cdots \;: repA = repB \;)
\end{array}
$$

What about uftype?

- implemented as a hash table
- mapping base terms to their parents
- should also store proofs

$$\text{uftype } =$$

$$\text{array} \; (\; \text{option} \; (\; base : \; T * parent : \; T * pf : base = parent \;))$$

Solution

Provide instantiation of free variables a term depends on, in the current environment

$$
\begin{array}{l}
\ldots \\
\forall x : Nat.P_1 \quad \mapsto \\
\quad \text{let } P_1' \qquad\qquad , \; pf' = \text{simplify } (\Phi, x : Nat) \; P_1 \\
\quad \text{in } (\; [\Phi](P_1' / (\Phi \mapsto id_\Phi, x \mapsto ??)) \; , \; \cdots \;)
\end{array}
$$

Solution

Provide instantiation of free variables a term depends on, in the current environment

$$\begin{array}{l} \ldots \\ \forall x : Nat.P_1 \quad \mapsto \\ \quad \text{let } P_1' : [\Phi, x : Nat]Prop \text{ , } pf' = \text{ simplify } (\Phi, x : Nat) \; P_1 \\ \quad \text{in } (\; [\Phi](P_1'/(\Phi \mapsto id_\Phi, x \mapsto ??)) \text{ , } \cdots \;) \end{array}$$