Towards Easier Reconstruction in Proof Assistants

Mathias Fleury

Isabelle at MDX – Summer School, 2023/06/27

Introduction

Feel free to interrupt me if you have questions!

Proofs from SMT Solvers

Use Cases

- Learning from proofs:
  - guidance: (FE)MaLeCoP, rlCoP (reinforcement learning), instance selection (veriT) …
- Unsatisfiable cores
- Finding interpolants
- Debugging
- Result certification if the problem is unsatisfiable

Proof Formats

Solver	Name	Experts	
Z3	Z3 proof format	1	Natural deduction, coarse, unmaintained de Moura&Bjørne, LPAR'08
veriT	old veriT format	3	Natural deduction [Besson et al., PxTP'11]
CVC4	LFSC	$\sim$ 6?	program that generates proofs no quantifiers [Stump et al., FMSD'13]
veriT	Alethe	3	FOL + choice [Schurr et al., PxTP'21]

Table 1: Different proof format

Proof Formats

Solver	Name	Experts	
Z3	Z3 proof format	1	Natural deduction, coarse, unmaintained de Moura&Bjørne, LPAR'08
veriT	old veriT format	3	Natural deduction [Besson et al., PxTP'11]
CVC4	LFSC	$\sim$ 6?	program that generates proofs no quantifiers [Stump et al., FMSD'13]
veriT	Alethe	3	FOL + choice [Schurr et al., PxTP'21]
cvc5	Alethe (ongoing)	$\sim$ 6?	FOL + choice

Table 1: Different proof format

Proof Formats

Solver	Name	Experts	
Z3	Z3 proof format	1	Natural deduction, coarse, unmaintained de Moura&Bjørne, LPAR'08
veriT	old veriT format	3	Natural deduction [Besson et al., PxTP'11]
CVC4	LFSC	$\sim$ 6?	program that generates proofs no quantifiers [Stump et al., FMSD'13]
veriT	Alethe	3	FOL + choice [Schurr et al., PxTP'21]
cvc5	Alethe (ongoing)	$\sim$ 6?	FOL + choice
Z3		1	Natural deduction [Z3 Github]

Table 1: Different proof format

Proof Formats

Solver	Name	Experts	
Z3	Z3 proof format	1	Natural deduction, coarse, unmaintained de Moura&Bjørne, LPAR'08
veriT	old veriT format	3	Natural deduction [Besson et al., PxTP'11]
CVC4	LFSC	$\sim$ 6?	program that generates proofs no quantifiers [Stump et al., FMSD'13]
veriT	Alethe	3	FOL + choice [Schurr et al., PxTP'21]
cvc5	Alethe (ongoing)	$\sim$ 6?	FOL + choice
Z3		1	Natural deduction [Z3 Github]
SMTInterpol		0	resolution-based [Hoenicke&Schindler, SMT'22]

Table 1: Different proof format

CVC4

```
(check
 ;; Declarations
(% b$ (term Bool)
(% f$ (term (arrow Bool Bool))
(% BOOLEAN_TERM_VARIABLE_231 (term Bool)
(% BOOLEAN_TERM_VARIABLE_233 (term Bool)
(% BOOLEAN_TERM_VARIABLE_235 (term Bool)
(% A1 (th_holds true)
(% A0 (th_holds (not (iff (p_app (apply _ _ f$ (f_to_b (p_app (apply _ _ f$ (f_to_b (p_app (apply _ _ f$ b
(: (holds cln)

[...]

 ;; Printing the global let map
[...]

 ;; In the preprocessor we trust
[...]

;; Printing mapping from preprocessed assertions into atoms
[...]
CVC4 suffered a segfault.
Offending address is 0x10
Looks like a NULL pointer was dereferenced.
```

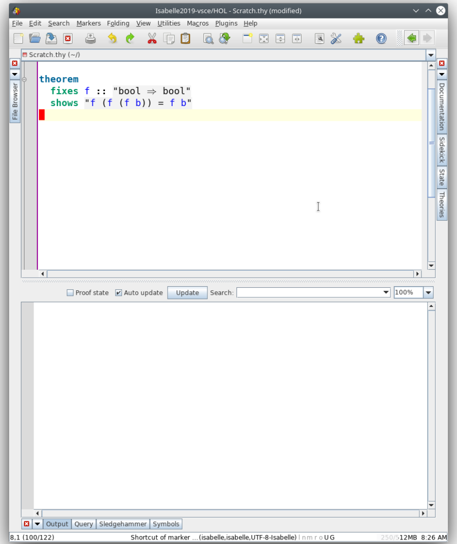
Z3 Proof

```
(proof
(let (($x28 (f$ b$)))
(let (($x29 (f$ $x28)))
(let (($x30 (f$ $x29)))
(let (($x40 (not $x30)))
(let (($x118 (= $x30 false)))
(let (($x49 (not $x28)))
(let (($x74 (= $x30 true)))
(let ((@x57 (hypothesis $x40)))
(let ((@x64 (iff-false (hypothesis (not $x29)) (= $x29 false))))
(let (($x41 (= $x40 $x28)))
(let ((@x39 (monotonicity (rewrite (= (= $x30 $x28) (= $x30 $x28))) (= (not (= $x30 $x28)) (not (= $x30 $x
(let ((@x45 (trans @x39 (rewrite (= (not (= $x30 $x28)) $x41)) (= (not (= $x30 $x28)) $x41))))
(let ((@x48 (mp (asserted (not (= $x30 $x28))) @x45 $x41)))
(let ((@x56 (unit-resolution (def-axiom (or $x30 $x28 (not $x41))) @x48 (or $x30 $x28))))
(let ((@x86 (symm (iff-true (unit-resolution @x56 @x57 $x28) (= $x28 true)) (= true $x28))))
(let (($x34 (= $x30 $x28)))
(let ((@x68 (symm (iff-false (hypothesis (not b$)) (= b$ false)) (= false b$))))
(let ((@x75 (trans (monotonicity (trans @x64 @x68 (= $x29 b$)) $x34) (iff-true (unit-resolution @x56 @x57
(let ((@x79 (unit-resolution @x57 (mp @x75 (rewrite (= $x74 $x30)) $x30) false)))
(let ((@x82 (unit-resolution (lemma @x79 (or b$ $x30 $x29)) (hypothesis (not $x29)) @x57 b$)))
(let ((@x90 (monotonicity (trans (iff-true @x82 (= b$ true)) @x86 (= b$ $x28)) (= $x28 $x29))))
(let ((@x97 (mp (trans (trans @x86 @x90 (= true $x29)) @x64 (= true false)) (rewrite (= (= true false) fal
(let ((@x102 (iff-true (unit-resolution (lemma @x97 (or $x29 $x30)) @x57 $x29) (= $x29 true))))
(let ((@x107 (trans (monotonicity (trans @x102 @x86 (= $x29 $x28)) (= $x30 $x29)) @x102 $x74)))
(let ((@x109 (unit-resolution @x57 (mp @x107 (rewrite (= $x74 $x30)) $x30) false)))
(let ((@x110 (lemma @x109 $x30)))
(let ((@x52 (unit-resolution (def-axiom (or $x40 $x49 (not $x41))) @x48 (or $x40 $x49))))
(let ((@x115 (symm (iff-false (unit-resolution @x52 @x110 $x49) (= $x28 false)) (= false $x28))))
```

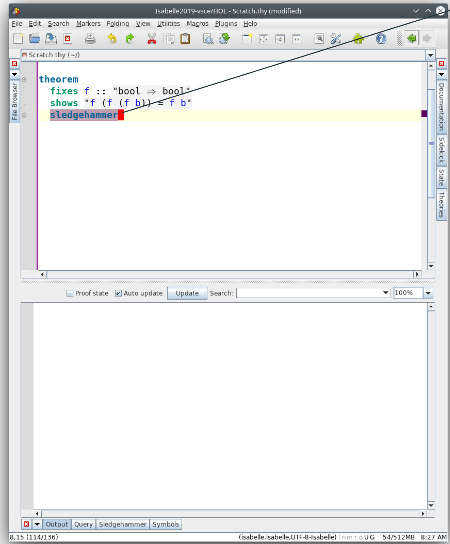
veriT Proof

```
(assume a0 (not (= (f$ (f$ (f$ b$))) (f$ b$))))
(step t2 (cl (not (= (ite (ite (ite b$ (f$ true) (f$ false)) (f$ true) (f$ false)) (f$ true) (f$ false)) (
(step t3 (cl (not (ite b$ (f$ true) (f$ false))) b$ (f$ false)) :rule ite_pos1)
(step t4 (cl (not (ite b$ (f$ true) (f$ false))) (not b$) (f$ true)) :rule ite_pos2)
(step t5 (cl (ite b$ (f$ true) (f$ false)) b$ (not (f$ false))) :rule ite_neg1)
(step t6 (cl (ite b$ (f$ true) (f$ false)) (not b$) (not (f$ true))) :rule ite_neg2)
(step t7 (cl (not (ite (ite b$ (f$ true) (f$ false)) (f$ true) (f$ false))) (ite b$ (f$ true) (f$ false))
(step t8 (cl (not (ite (ite b$ (f$ true) (f$ false)) (f$ true) (f$ false))) (not (ite b$ (f$ true) (f$ fal
(step t9 (cl (ite (ite b$ (f$ true) (f$ false)) (f$ true) (f$ false)) (ite b$ (f$ true) (f$ false)) (not (
(step t10 (cl (ite (ite b$ (f$ true) (f$ false)) (f$ true) (f$ false)) (not (ite b$ (f$ true) (f$ false))
(step t11 (cl (not (ite (ite (ite b$ (f$ true) (f$ false)) (f$ true) (f$ false)) (f$ true) (f$ false))) (i
(step t12 (cl (not (ite (ite (ite b$ (f$ true) (f$ false)) (f$ true) (f$ false)) (f$ true) (f$ false))) (n
(step t13 (cl (ite (ite (ite b$ (f$ true) (f$ false)) (f$ true) (f$ false)) (f$ true) (f$ false)) (ite (it
(step t14 (cl (ite (ite (ite b$ (f$ true) (f$ false)) (f$ true) (f$ false)) (f$ true) (f$ false)) (not (it
(step t15 (cl (ite (ite (ite b$ (f$ true) (f$ false)) (f$ true) (f$ false)) (f$ true) (f$ false)) (ite b$
(step t16 (cl (not (ite (ite (ite b$ (f$ true) (f$ false)) (f$ true) (f$ false)) (f$ true) (f$ false))) (n
(step t17 (cl (ite (ite (ite b$ (f$ true) (f$ false)) (f$ true) (f$ false)) (f$ true) (f$ false)) (not (it
(step t18 (cl (ite (ite (ite b$ (f$ true) (f$ false)) (f$ true) (f$ false)) (f$ true) (f$ false))) :rule r
(step t19 (cl (not (ite b$ (f$ true) (f$ false)))) :rule resolution :premises (t16 t18))
(step t20 (cl (ite (ite b$ (f$ true) (f$ false)) (f$ true) (f$ false))) :rule resolution :premises (t11 t9
(step t21 (cl (f$ false)) :rule resolution :premises (t7 t20 t19))
(step t22 (cl (f$ true)) :rule resolution :premises (t12 t20 t18))
(step t23 (cl b$) :rule resolution :premises (t5 t21 t19))
(step t24 (cl) :rule resolution :premises (t6 t22 t19 t23))
```

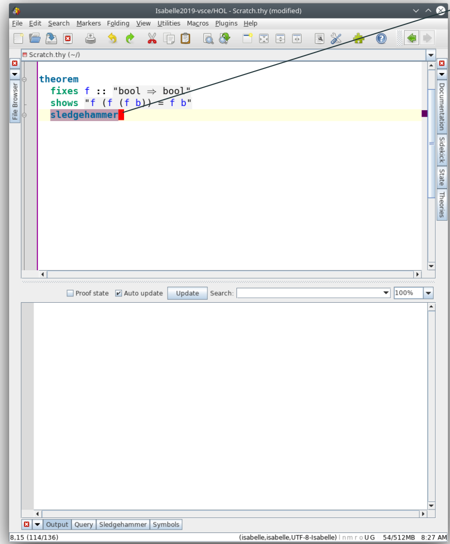
Isabelle and Reconstruction Kaminski’s Theorem



Fact filtering



Isabelle and Reconstruction Kaminski's Theorem



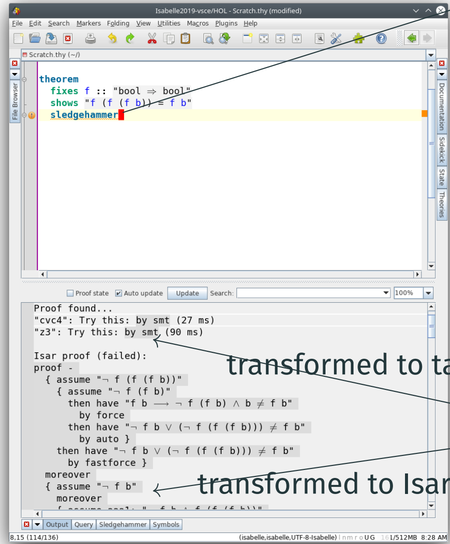
Fact filtering

ATPs

CVC4, E, SPASS, Vampire, Z3

Proof or UNSAT Core

Fact filtering



ATPs

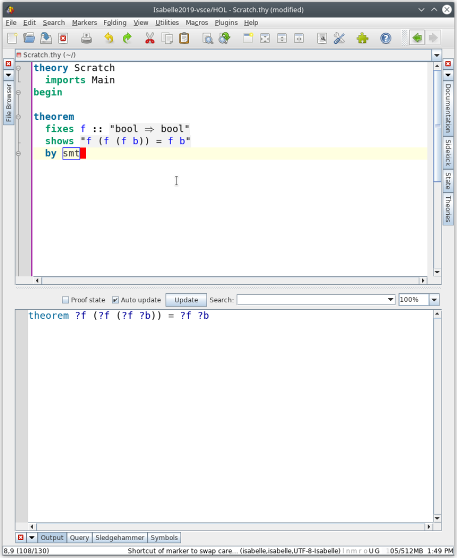
CVC4, E, SPASS, Vampire, Z3

transformed to tactic

Proof or UN-SAT Core

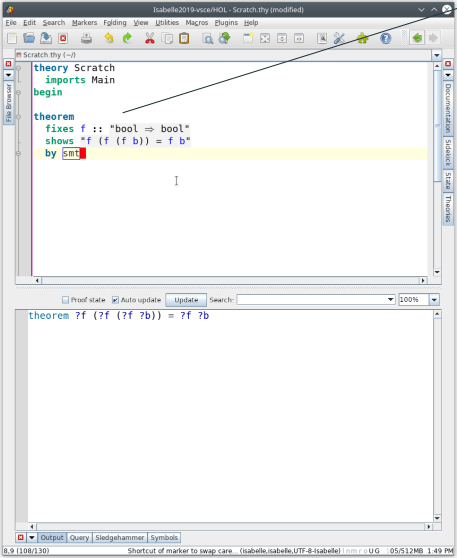
transformed to Isar script

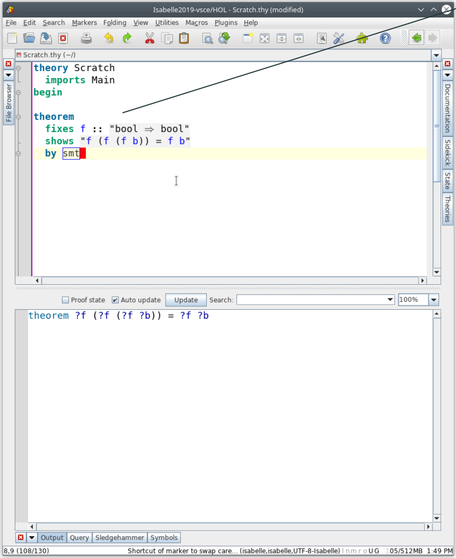
Isabelle and Reconstruction Kaminski's Theorem



Isabelle and Reconstruction Kaminski's Theorem

~~Fact filtering~~





~~Fact filtering~~

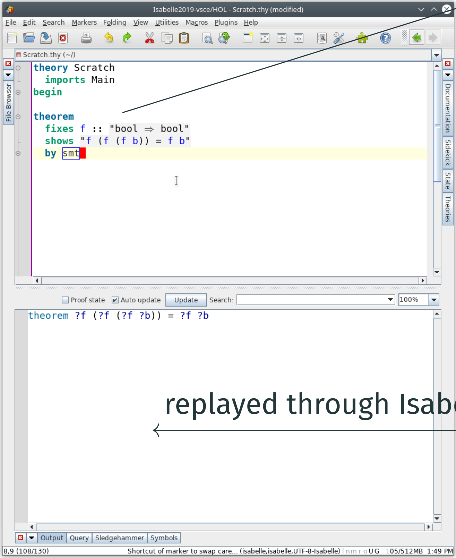
~~ATPs~~

~~CVC4, E, SPASS, Vampire,~~ Z3

Proof

Isabelle and Reconstruction Kaminski's Theorem

~~Fact filtering~~



~~ATPs~~

~~CVC4, E, SPASS, Vampire,~~ Z3

replayed through Isabelle core

Proof

Challenges for proofs in FOL

1. Collecting and storing proof information efficiently
    - no convergence, but quite active
2. Proofs for sophisticated processing techniques
    - proof with holes or too coarse
3. Producing proofs for modules that use external tools
    - tool dependent
4. Standard proof format
    - open

Challenges for proofs in FOL

1. Collecting and storing proof information efficiently
    - no convergence, but quite active
2. Proofs for sophisticated processing techniques
    - ~~proof with holes or too coarse~~Scalable fine-grained proofs
3. Producing proofs for modules that use external tools
    - tool dependent
4. Standard proof format
    - open

Challenges for proofs in FOL

1. Collecting and storing proof information efficiently
    - no convergence, but quite active
2. Proofs for sophisticated processing techniques
    - ~~proof with holes or too coarse~~Scalable fine-grained proofs
3. Producing proofs for modules that use external tools
    - tool dependent
4. Standard proof format
    - open

Success story: DRAT for SAT solvers; less succesful: TPTP for superposition provers

Challenges for proofs in FOL

1. Collecting and storing proof information efficiently
   - no convergence, but quite active
2. Proofs for sophisticated processing techniques
   - ~~proof with holes or too coarse~~Scalable fine-grained proofs
3. Producing proofs for modules that use external tools
   - tool dependent
4. Standard proof format
   - open

Success story: DRAT for SAT solvers; less succesful: TPTP for superposition provers

DRAT: generated on the fly, and not at the *end*

What do We do With Proofs? SMT solvers want to generate proofs and have easy problems.

We want good proofs!

Proof Format

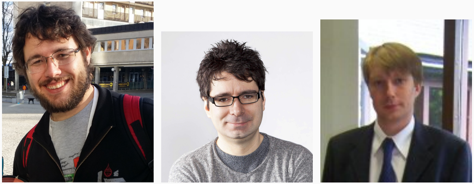


Figure 1: Haniel Barbosa, Jasmin Blanchette, and Pascal Fontaine

Based on the 2017 CADE slide by Haniel Barbosa.

The SMT Solver veriT

- Traditional CDCL(T) solver
- Supports:
  - Uninterpreted functions
  - Linear Arithmetic
  - Non-Linear Arithmetic
  - Quantifiers
  - ...
- Proof producing
- SMT-LIB input

-S trac -f latex -P commandprefix=PYG-S trac -f latex -P commandprefix=PYG

```
(set-option :produce-proofs true)
(set-logic AUFLIA)
(declare-sort A$ 0)
(declare-sort A_list$ 0)
(declare-fun p$ (A_list$) Bool)
(declare-fun x1$ () A_list$)
(declare-fun x2$ () A$)
(declare-fun ys$ () A_list$)
(declare-fun xs2$ () A_list$)
(declare-fun cons$ (A$ A_list$) A_list$)
(declare-fun append$ (A_list$ A_list$) A_list$)
(assert (! (forall ((?v0 A_list$) (?v1 A_list$)
 (?v2 A_list$)) (= (append$ (append$ ?v0 ?v1) ?v2)
 (append$ ?v0 (append$ ?v1 ?v2)))) :named a0))
(assert (! (forall ((?v0 A_list$) (?v1 A$)
 (?v2 A_list$)) (=> (= (append$ ?v0 (cons$ ?v1 ?v2))
 (append$ x1$ (append$ xs2$ (cons$ x2$ ys$))))
 (p$ ys$))) :named a1))
(assert (! (not (p$ ys$)) :named a2))
(check-sat)
(get-proof)
```

What is hard?

SAT Solver Resolution, $A \vee \ell$ and $B \vee \neg\ell$ implies $A \vee B$

Theory solvers lemmas like $x < y \vee x > y \vee x = y$ or
$\neg(x = y) \vee f(x) = f(y)$

Instantiation Module lemmas like $\neg(\forall x.\ \phi[x]) \vee \phi[t]$

What is hard?

SAT Solver Resolution, $A \vee \ell$ and $B \vee \neg\ell$ implies $A \vee B$

- preserves logical equivalence but not all transformations do

Theory solvers lemmas like $x < y \vee x > y \vee x = y$ or $\neg(x = y) \vee f(x) = f(y)$

- not detailed enough for simplifications

Instantiation Module lemmas like $\neg(\forall x.\ \phi[x]) \vee \phi[t]$

- Complicated code, many cases

Proofs in veriT

Idea: *local* transformations with binders.

Skolemization $\neg(\forall x.\ \phi(x)) \simeq \neg p(\varepsilon x.\ \neg\phi(x))$

Let-elim $(\text{let } x = a \text{ in } p(x,x)) \simeq p(a,a)$

Theory simplification $k + 1 \times 0 < k \simeq k < k$

Challenge: efficient and sound manipulation of bound variables

Definition

Rules have the form

premises

$$\frac{\mathcal{D}_1 \qquad \cdots \qquad \mathcal{D}_n}{\Gamma \;\triangleright\; t \simeq u}\,\mathsf{R}$$

assumptions Transformation

Semantics: proof of $\Gamma(t) = u$ for all variables fixed by Γ

Definition

A context Γ fixes variables and specifies substitutions:

$$\Gamma ::= \emptyset \mid \Gamma, x \mapsto s \mid \Gamma, x$$

substitution bound variable

Rules have the form

premises

$$\frac{\mathcal{D}_1 \qquad \cdots \qquad \mathcal{D}_n}{\Gamma \;\triangleright\; t \simeq u}\ \mathrm{R}$$

assumptions Transformation

Semantics: proof of $\Gamma(t) = u$ for all variables fixed by Γ

Alethe's Proofs

-S trac -f latex -P commandprefix=PYG

```
(assume h1 (not (p a)))
(assume h2 (forall ((z1 U)) (forall ((z2 U)) (p z2))))
...
(anchor :step t9 :args ((:= z2 veriT_vr4)))
(step t9.t1 (cl (= z2 veriT_vr4)) :rule refl)
(step t9.t2 (cl (= (p z2) (p veriT_vr4))) :rule cong :premises (t9.t1))
(step t9 (cl (= (forall ((z2 U)) (p z2))
                (forall ((veriT_vr4 U)) (p veriT_vr4)))) :rule bind)
...
(step t14 (cl (forall ((veriT_vr5 U)) (p veriT_vr5)))
          :rule th_resolution :premises (t11 t12 t13))
(step t15 (cl (or (not (forall ((veriT_vr5 U)) (p veriT_vr5))) (p a)))
          :rule forall_inst :args ((:= veriT_vr5 a)))
(step t16 (cl (not (forall ((veriT_vr5 U)) (p veriT_vr5))) (p a))
          :rule or :premises (t15))
(step t17 (cl) :rule resolution :premises (t16 h1 t14))
```

Alethe's Proofs

-S trac -f latex -P commandprefix=PYG

```
(assume h1 (not (p a)))
(assume h2 (forall ((z1 U)) (forall ((z2 U)) (p z2))))
...
(anchor :step t9 :args ((:= z2 veriT_vr4)))
(step t9.t1 (cl (= z2 veriT_vr4)) :rule refl)
(step t9.t2 (cl (= (p z2) (p veriT_vr4))) :rule cong :premises (t9.t1))
(step t9 (cl (= (forall ((z2 U)) (p z2))
                (forall ((veriT_vr4 U)) (p veriT_vr4)))) :rule bind)
...
(step t14 (cl (forall ((veriT_vr5 U)) (p veriT_vr5)))
          :rule th_resolution :premises (t11 t12 t13))
(step t15 (cl (or (not (forall ((veriT_vr5 U)) (p veriT_vr5))) (p a)))
          :rule forall_inst :args ((:= veriT_vr5 a)))
(step t16 (cl (not (forall ((veriT_vr5 U)) (p veriT_vr5))) (p a))
          :rule or :premises (t15))
(step t17 (cl) :rule resolution :premises (t16 h1 t14))
```

Input assumptions

Alethe's Proofs

-S trac -f latex -P commandprefix=PYG

Simple step

```
(assume h1 (not (p a)))
(assume h2 (forall ((z1 U)) (forall ((z2 U)) (p z2))))
...
(anchor :step t9 :args ((:= z2 veriT_vr4)))
(step t9.t1 (cl (= z2 veriT_vr4)) :rule refl)
(step t9.t2 (cl (= (p z2) (p veriT_vr4))) :rule cong :premises (t9.t1))
(step t9 (cl (= (forall ((z2 U)) (p z2))
                (forall ((veriT_vr4 U)) (p veriT_vr4)))) :rule bind)
...
(step t14 (cl (forall ((veriT_vr5 U)) (p veriT_vr5)))
          :rule th_resolution :premises (t11 t12 t13))
(step t15 (cl (or (not (forall ((veriT_vr5 U)) (p veriT_vr5))) (p a)))
          :rule forall_inst :args ((:= veriT_vr5 a)))
(step t16 (cl (not (forall ((veriT_vr5 U)) (p veriT_vr5))) (p a))
          :rule or :premises (t15))
(step t17 (cl) :rule resolution :premises (t16 h1 t14))
```

Alethe's Proofs

-S trac -f latex -P commandprefix=PYG

```
(assume h1 (not (p a)))
(assume h2 (forall ((z1 U)) (forall ((z2 U)) (p z2))))
...
(anchor :step t9 :args ((:= z2 veriT_vr4)))
(step t9.t1 (cl (= z2 veriT_vr4)) :rule refl)
(step t9.t2 (cl (= (p z2) (p veriT_vr4))) :rule cong :premises (t9.t1))
(step t9 (cl (= (forall ((z2 U)) (p z2))
                (forall ((veriT_vr4 U)) (p veriT[illegible]ule bind)
...
(step t14 (cl (forall ((veriT_vr5 U)) (p veriT_vr5)))
          :rule th_resolution :premises (t11 t12 t13))
(step t15 (cl (or (not (forall ((veriT_vr5 U)) (p veriT_vr5))) (p a)))
          :rule forall_inst :args ((:= veriT_vr5 a)))
(step t16 (cl (not (forall ((veriT_vr5 U)) (p veriT_vr5))) (p a))
          :rule or :premises (t15))
(step t17 (cl) :rule resolution :premises (t16 h1 t14))
```

Name

Alethe's Proofs

-S trac -f latex -P commandprefix=PYG

```
(assume h1 (not (p a)))
(assume h2 (forall ((z1 U)) (forall ((z2 U)) (p z2))))
...
(anchor :step t9 :args ((:= z2 veriT_vr4)))
(step t9.t1 (cl (= z2 veriT_vr4)) :rule refl)
(step t9.t2 (cl (= (p z2) (p veriT_vr4))) :rule cong :premises (t9.t1))
(step t9 (cl (= (forall ((z2 U)) (p z2))
                (forall ((veriT_vr4 U)) (p veriT_vr4)))) :rule bind)
...
(step t14 (cl (forall ((veriT_vr5 U)) (p veriT_vr5)))
          :rule th_resolution :premises (t11 t12 t13))
(step t15 (cl (or (not (forall ((veriT_vr5 U)) (p veriT_vr5))) (p a)))
          :rule forall_inst :args ((:= veriT_vr5 a)))
(step t16 (cl (not (forall ((veriT_vr5 U)) (p veriT_vr5))) (p a))
          :rule or :premises (t15))
(step t17 (cl) :rule resolution :premises (t16 h1 t14))
```

Introduced term

Alethe's Proofs

-S trac -f latex -P commandprefix=PYG

```
(assume h1 (not (p a)))
(assume h2 (forall ((z1 U)) (forall ((z2 U)) (p z2))))
...
(anchor :step t9 :args ((:= z2 veriT_vr4)))
(step t9.t1 (cl (= z2 veriT_vr4)) :rule refl)
(step t9.t2 (cl (= (p z2) (p veriT_vr4))) :rule cong :premises (t9.t1))
(step t9 (cl (= (forall ((z2 U)) (p z2))
                (forall ((veriT_vr4 U)) (p veriT_vr4)))) :rule bind)
...
(step t14 (cl (forall ((veriT_vr5 U)) (p veriT_vr5)))
          :rule th_resolution :premises (t11 t12 t13))
(step t15 (cl (or (not (forall ((veriT_vr5 U)) (p ver  (p a)))
          :rule forall_inst :args ((:= veriT_vr5 a)))
(step t16 (cl (not (forall ((veriT_vr5 U)) (p veriT_vr5))) (p a))
          :rule or :premises (t15))
(step t17 (cl) :rule resolution :premises (t16 h1 t14))
```

Rule

Alethe's Proofs

-S trac -f latex -P commandprefix=PYG

```
(assume h1 (not (p a)))
(assume h2 (forall ((z1 U)) (forall ((z2 U)) (p z2))))
...
(anchor :step t9 :args ((:= z2 veriT_vr4)))
(step t9.t1 (cl (= z2 veriT_vr4)) :rule refl)
(step t9.t2 (cl (= (p z2) (p veriT_vr4))) :rule cong :premises (t9.t1))
(step t9 (cl (= (forall ((z2 U)) (p z2))
                (forall ((veriT_vr4 U)) (p veriT_vr4)))) :rule bind)
...
(step t14 (cl (forall ((veriT_vr5 U)) (p veriT_vr5)))
          :rule th_resolution :premises (t11 t12 t13))
(step t15 (cl (or (not (forall ((veriT_vr5 U)) (p veriT_vr5))) (p a)))
          :rule forall_inst :args ((:= veriT_vr5 a
(step t16 (cl (not (forall ((veriT_vr5 U)) (p veri
          :rule or :premises (t15))
(step t17 (cl) :rule resolution :premises (t16 h1 t14))
```

Premises

Alethe's Proofs

-S trac -f latex -P commandprefix=PYG

```
(assume h1 (not (p a)))
(assume h2 (forall ((z1 U)) (forall ((z2 U)) (p z2))))
...
(anchor :step t9 :args ((:= z2 veriT_vr4)))
(step t9.t1 (cl (= z2 veriT_vr4)) :rule refl)
(step t9.t2 (cl (= (p z2) (p veriT_vr4))) :rule cong :premises (t9.t1))
(step t9 (cl (= (forall ((z2 U)) (p z2))
                (forall ((veriT_vr4 U)) (p veriT_vr4)))) :rule bind)
...
(step t14 (cl (forall ((veriT_vr5 U)) (p veriT_vr5)))
          :rule th_resolution :premises (t11 t12 t13))
(step t15 (cl (or (not (forall ((veriT_vr5 U)) (p veriT_vr5))) (p a)))
          :rule forall_inst :args ((:= veriT_vr5 a)))
(step t16 (cl (not (forall ((veriT_vr5 U)) (p veriT_vr5))) (p a))
          :rule or :premises (t15))
(step t17 (cl) :rule resolution :premises (t16 h1 t14))
```

Context annotation

Demo-Time

```
lemma
  assumes <!x. P x> and <¬P x>
  shows False
  using assms
  supply [[smt_trace,verit_compress_proofs]]
  apply (smt (verit))
  oops
```

Proof-producing contextual recursion

$$
\begin{array}{l}
\textbf{\textit{function}}\ \textit{process}(\Gamma, t) \\
\quad \textbf{\textit{match}}\ t \\
\quad\quad \textbf{\textit{case}}\ x\text{:} \\
\quad\quad\quad \textbf{\textit{return}}\ \textit{build\_var}(\Gamma, x) \\
\quad\quad \textbf{\textit{case}}\ \mathsf{f}(\bar{t}_n)\text{:} \\
\quad\quad\quad \bar{\Gamma}'_n \leftarrow (\textit{ctx\_app}(\Gamma, \mathsf{f}, \bar{t}_n, i))_{i=1}^{n} \\
\quad\quad\quad \textbf{\textit{return}}\ \textit{build\_app}\big(\Gamma, \bar{\Gamma}'_n, \mathsf{f}, \bar{t}_n, (\textit{process}(\Gamma'_i, t_i))_{i=1}^{n}\big) \\
\quad\quad \textbf{\textit{case}}\ Qx.\,\varphi\text{:} \\
\quad\quad\quad \Gamma' \leftarrow \textit{ctx\_quant}(\Gamma, Q, x, \varphi) \\
\quad\quad\quad \textbf{\textit{return}}\ \textit{build\_quant}(\Gamma, \Gamma', Q, x, \varphi, \textit{process}(\Gamma', \varphi)) \\
\quad\quad \textbf{\textit{case}}\ \mathrm{let}\ \bar{x}_n \simeq \bar{r}_n\ \mathrm{in}\ t'\text{:} \\
\quad\quad\quad \Gamma' \leftarrow \textit{ctx\_let}(\Gamma, \bar{x}_n, \bar{r}_n, t') \\
\quad\quad\quad \textbf{\textit{return}}\ \textit{build\_let}(\Gamma, \Gamma', \bar{x}_n, \bar{r}_n, t', \textit{process}(\Gamma', t'))
\end{array}
$$

Theoretical Properties

Soundness of inference rules proven through an encoding into simply typed logic

Theorem

Soundness If $\Gamma \;\triangleright\; t \simeq u$ *is derivable using* $\mathcal{T} = \mathcal{T}_1 \cup \cdots \cup \mathcal{T}_n \cup \sim \cup \mathtt{let}$, *then* $\models_{\mathcal{T}} \Gamma(t) \sim u$.

Correctness of proof-producing contextual recursion algorithm

Cost of proof production is linear

Cost of proof checking is almost linear with the suitable data structure and maximal sharing inference rules involve shallow conditions on contexts and terms

Theoretical Properties

Soundness of inference rules proven through an encoding into simply typed logic

Theorem

Soundness If $\Gamma \ \triangleright \ t \simeq u$ *is derivable using* $\mathcal{T} = \mathcal{T}_1 \cup \cdots \cup \mathcal{T}_n \cup \sim \cup \mathtt{let}$, *then* $\models_{\mathcal{T}} \Gamma(t) \sim u$.

Correctness of proof-producing contextual recursion algorithm

Cost of proof production is linear

Cost of proof checking is almost linear with the suitable data structure and maximal sharing [1] inference rules involve shallow conditions on contexts and terms

[1] which (probably) no proof assistant has

Implementation

Proof output for veriT

- fine-grained for most processing transformation
- no negative impact on performance
- more transformations used in proof producing mode
- simplification of the code base

Proof Reconstruction

Figure 2: Hans-Jörg Schurr

Based on the 2019 AITP slide by Hans-Jörg Schurr.

Setting Sails

Collaborate

Given that we are both developers of the SMT solver and the reconstruction, many problems (bugs, unclarities, etc.) can be solved on short notice.

Documentation

- Automatically generated: `--proof-format-and-exit`
  - Necessarily contains all rules...
  - ... but not necessary a description

Weight: Proof Size

Problem Proofs are often huge

- choice terms introduced by skolemization can be huge
- 62MB proof: not parsable by Isabelle

Solution Sharing of terms

- `(!` t `:named` n`)` syntax of SMT-LIB
- 192KB proof, now parsable

Proof Without Sharing

-S trac -f latex -P commandprefix=PYG

```
(assume h1 (and (forall ((?veriT.veriT__4 Client) (?veriT.veriT__3 Client)) (= ?veriT.veriT__4 ?veriT.veri
c1 c2))))
(anchor :step t2 :args ((:= ?veriT.veriT__4 veriT_vr0) (:= ?veriT.veriT__3 veriT_vr1)))
(step t2.t1 (cl (= ?veriT.veriT__4 veriT_vr0)) :rule refl)
(step t2.t2 (cl (= ?veriT.veriT__3 veriT_vr1)) :rule refl)
(step t2.t3 (cl (= (= ?veriT.veriT__4 ?veriT.veriT__3) (= veriT_vr0 veriT_vr1))) :rule cong :premises (t2.
(step t2 (cl (= (forall ((?veriT.veriT__4 Client) (?veriT.veriT__3 Client)) (= ?veriT.veriT__4 ?veriT.veri
((veriT_vr0 Client) (veriT_vr1 Client)) (= veriT_vr0 veriT_vr1)))) :rule bind)
(step t3 (cl (= (and (forall ((?veriT.veriT__4 Client) (?veriT.veriT__3 Client)) (= ?veriT.veriT__4 ?veriT
(not (= c1 c2))) (and (forall ((veriT_vr0 Client) (veriT_vr1 Client)) (= veriT_vr0 veriT_vr1)) (not (= c1
cong :premises (t2))
(step t4 (cl (not (= (and (forall ((?veriT.veriT__4 Client) (?veriT.veriT__3 Client)) (= ?veriT.veriT__4 ?
(not (= c1 c2))) (and (forall ((veriT_vr0 Client) (veriT_vr1 Client)) (= veriT_vr0 veriT_vr1)) (not (= c1
(and (forall ((?veriT.veriT__4 Client) (?veriT.veriT__3 Client)) (= ?veriT.veriT__4 ?veriT.veriT__3)) (no
(and (forall ((veriT_vr0 Client) (veriT_vr1 Client)) (= veriT_vr0 veriT_vr1)) (not (= c1 c2)))) :rule equ
(step t5 (cl (and (forall ((veriT_vr0 Client) (veriT_vr1 Client)) (= veriT_vr0 veriT_vr1)) (not (= c1 c2))
th_resolution :premises (h1 t3 t4))
(anchor :step t6 :args ((:= veriT_vr0 veriT_vr2) (:= veriT_vr1 veriT_vr3)))
(step t6.t1 (cl (= veriT_vr0 veriT_vr2)) :rule refl)
(step t6.t2 (cl (= veriT_vr1 veriT_vr3)) :rule refl)
(step t6.t3 (cl (= (= veriT_vr0 veriT_vr1) (= veriT_vr2 veriT_vr3))) :rule cong :premises (t6.t1 t6.t2))
(step t6 (cl (= (forall ((veriT_vr0 Client) (veriT_vr1 Client)) (= veriT_vr0 veriT_vr1)) (forall ((veriT_v
(veriT_vr3 Client)) (= veriT_vr2 veriT_vr3)))) :rule bind)
(step t7 (cl (= (and (forall ((veriT_vr0 Client) (veriT_vr1 Client)) (= veriT_vr0 veriT_vr1)) (not (= c1 c
((veriT_vr2 Client) (veriT_vr3 Client)) (= veriT_vr2 veriT_vr3)) (not (= c1 c2))))) :rule cong :premises (
(step t8 (cl (not (= (and (forall ((veriT_vr0 Client) (veriT_vr1 Client)) (= veriT_vr0 veriT_vr1)) (not (=
(forall ((veriT_vr2 Client) (veriT_vr3 Client)) (= veriT_vr2 veriT_vr3)) (not (= c1 c2))))) (not (and (for
```

Proof With Sharing

-S trac -f latex -P commandprefix=PYG

```
(assume h1 (! (and (! (forall ((?veriT.veriT__4 Client) (?veriT.veriT__3 Client)) (! (= ?veriT.veriT__4 ?v
:named @p_3)) :named @p_2) (! (not (! (= c1 c2) :named @p_5)) :named @p_4)) :named @p_1))
(anchor :step t2 :args ((:= ?veriT.veriT__4 veriT_vr0) (:= ?veriT.veriT__3 veriT_vr1)))
(step t2.t1 (cl (! (= ?veriT.veriT__4 veriT_vr0) :named @p_6)) :rule refl)
(step t2.t2 (cl (! (= ?veriT.veriT__3 veriT_vr1) :named @p_7)) :rule refl)
(step t2.t3 (cl (! (= @p_3 (! (= veriT_vr0 veriT_vr1) :named @p_9)) :named @p_8)) :rule cong :premises (t2
(step t2 (cl (! (= @p_2 (! (forall ((veriT_vr0 Client) (veriT_vr1 Client)) @p_9) :named @p_11)) :named @p_
(step t3 (cl (! (= @p_1 (! (and @p_11 @p_4) :named @p_13)) :named @p_12)) :rule cong :premises (t2))
(step t4 (cl (! (not @p_12) :named @p_14) (! (not @p_1) :named @p_15) @p_13) :rule equiv_pos2)
(step t5 (cl @p_13) :rule th_resolution :premises (h1 t3 t4))
(anchor :step t6 :args ((:= veriT_vr0 veriT_vr2) (:= veriT_vr1 veriT_vr3)))
(step t6.t1 (cl (! (= veriT_vr0 veriT_vr2) :named @p_16)) :rule refl)
(step t6.t2 (cl (! (= veriT_vr1 veriT_vr3) :named @p_17)) :rule refl)
(step t6.t3 (cl (! (= @p_9 (! (= veriT_vr2 veriT_vr3) :named @p_19)) :named @p_18)) :rule cong :premises (
(step t6 (cl (! (= @p_11 (! (forall ((veriT_vr2 Client) (veriT_vr3 Client)) @p_19) :named @p_21)) :named @
(step t7 (cl (! (= @p_13 (! (and @p_21 @p_4) :named @p_23)) :named @p_22)) :rule cong :premises (t6))
(step t8 (cl (! (not @p_22) :named @p_24) (! (not @p_13) :named @p_25) @p_23) :rule equiv_pos2)
(step t9 (cl @p_23) :rule th_resolution :premises (t5 t7 t8))
(step t10 (cl @p_21) :rule and :premises (t9))
(step t11 (cl @p_4) :rule and :premises (t9))
(step t12 (cl (! (or (! (not @p_21) :named @p_27) @p_5) :named @p_26)) :rule forall_inst :args ((:= veriT_
veriT_vr3 c1)))
(step t13 (cl @p_27 @p_5) :rule or :premises (t12))
(step t14 (cl) :rule resolution :premises (t13 t10 t11))
```

In Practice

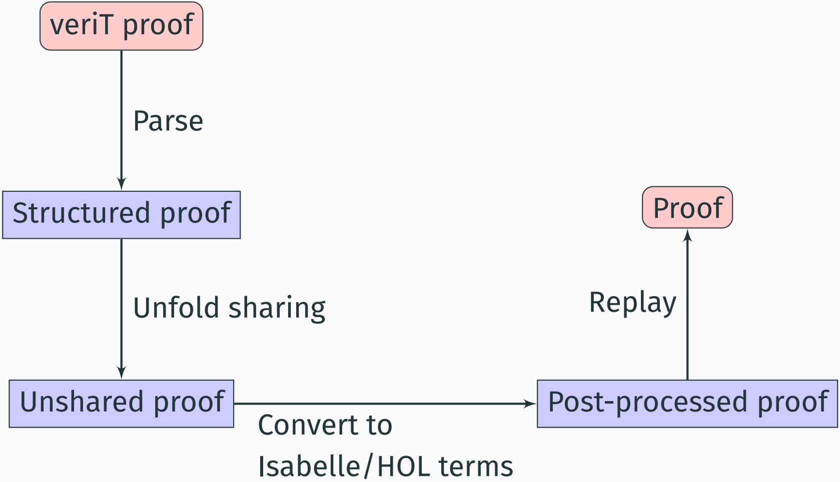
Where to introduce names?

- Perfect solution is hard to find
- Approximate: Terms which appear with two different parents get a name
  - $f(h(a), j(x, y)), g(h(a)), g(f(h(a), j(x, y)))$
  - $[f([h(a)]_{p_2}, j(x, y))]_{p_1}, [g(p_2)]_{p_3}, [g(p_1)]_{p_4}$
- Can be done in linear time thanks to perfect sharing

Isabelle/HOL side

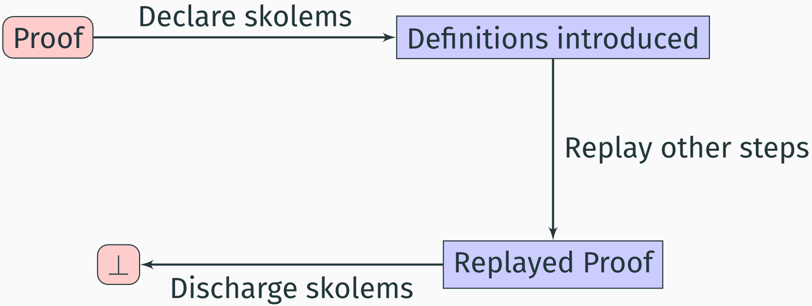
- Isabelle/HOL unfolds everything
- (previously: ...except for skolem terms)

The Reconstruction Inside Isabelle/HOL



The Reconstruction Inside Isabelle/HOL

$\mathtt{sk}_1$



$\bot$

$\mathtt{sk}_1 = \varepsilon x.\ \cdots \implies \textit{False}$

$\forall \mathtt{sk}_1.\ (\mathtt{sk}_1 = \varepsilon x.\ \cdots \implies \bot)$

Reconstruction

`* equiv1 :` $(a \iff b) \Rightarrow \neg a \vee b$

Reconstruction

Direct Proof Rules

- Proof of B assuming A
- Semantics: rule $A \Rightarrow B'$
- We assume A
- We derive B'
- then `simp/fast/blast` to discharge $B' \Rightarrow B$

$(\neg a \iff a) \Rightarrow a$

Challenges

- Implicit steps
  - Order of $=$ is freely changed
  - Step simplification: $a \simeq b \wedge a \simeq b \Rightarrow f(a,a) \simeq f(b,b)$ $a \simeq b \Rightarrow f(a,a) \simeq f(b,b)$
  - Double negation is eliminated

Reconstruction

Direct Proof Rules

- Proof of B assuming A
- Semantics: rule $A \Rightarrow B'$
- We assume A
- We derive B'
- then `simp/fast/blast` to discharge $B' \Rightarrow B$

The order of $=$ is not random, so we are relying on implementation details

Challenges

- Implicit steps
  - Order of $=$ is freely changed
  - Step simplification: $a \simeq b \wedge a \simeq b \Rightarrow f(a,a) \simeq f(b,b)$ $a \simeq b \Rightarrow f(a,a) \simeq f(b,b)$
  - Double negation is eliminated

Reconstruction

Direct Proof Rules

- Proof of B assuming A
- Semantics: rule $A \Rightarrow B'$
- We assume A
- We derive B'
- then `simp/fast/blast` to discharge $B' \Rightarrow B$

The order of $=$ is not random, so we are relying on implementation details

Challenges

- Implicit steps
  - Order of $=$ is freely changed
  - Step simplification: $a \simeq b \wedge a \simeq b \Rightarrow f(a,a) \simeq f(b,b)$ $a \simeq b \Rightarrow f(a,a) \simeq f(b,b)$
  - Double negation is eliminated
- Skolemization

At the beginning everything was fine and veriT produced the step:

$$\forall x.p[x] \rightarrow p[t]$$

At the beginning everything was fine and veriT produced the step:

$$\forall x. p[x] \to p[t]$$

Then: ≪If we have $\forall x.\,(p_1 \land p_2 \land p_3)$ we can produce $\forall x.\,(p_1 \land p_2 \land p_3) \to p_i[t]$.≫

- Only a few lines of code change
- This change was done a while ago
- Without reconstruction we would never have known

At the beginning everything was fine and veriT produced the step:

$$\forall x.p[x] \rightarrow p[t]$$

Then: ≪If we have $\forall x.\ (p_1 \wedge p_2 \wedge p_3)$ we can produce $\forall x.\ (p_1 \wedge p_2 \wedge p_3) \rightarrow p_i[t]$.≫

- Only a few lines of code change
- This change was done a while ago
- Without reconstruction we would never have known

Since then: Under some circumstances $p[x]$ is a CNF of another formula.

- Reconstruction forces you to stay honest

Current State: Works Great!

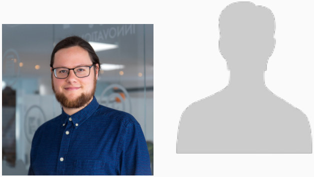
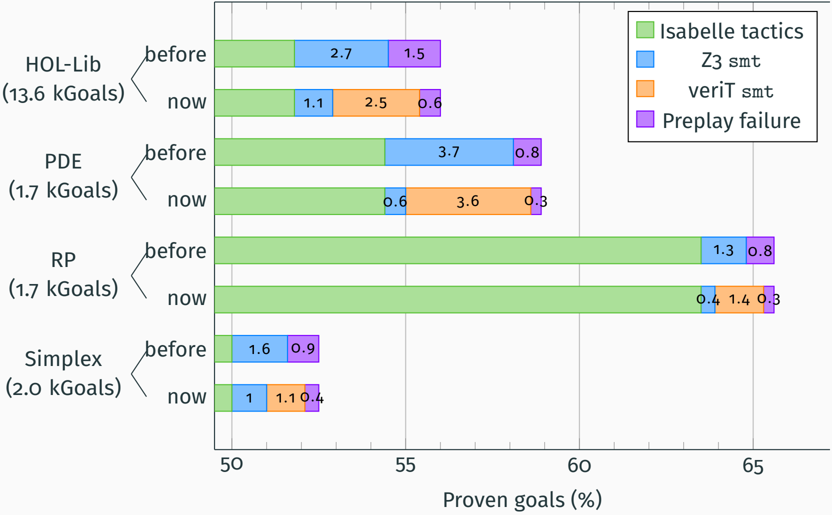


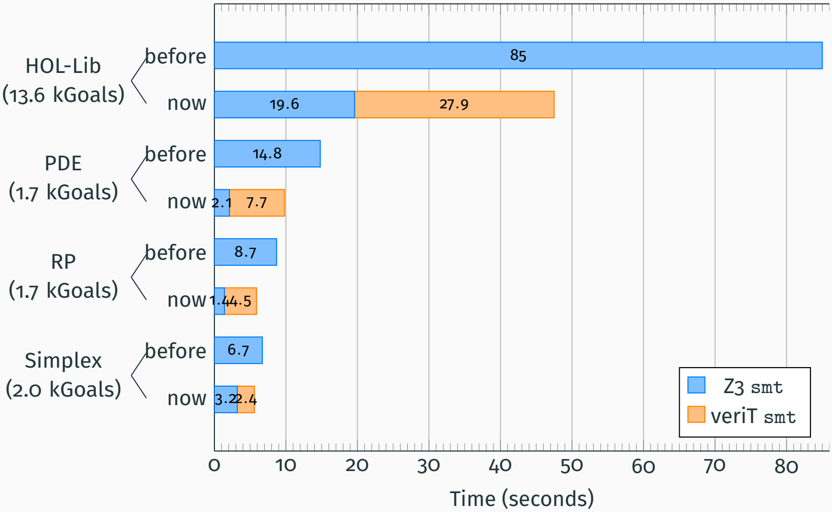
Figure 3: Hans-Jörg Schurr and Martin Desharnais

Based on the 2023 CADE 28 slide

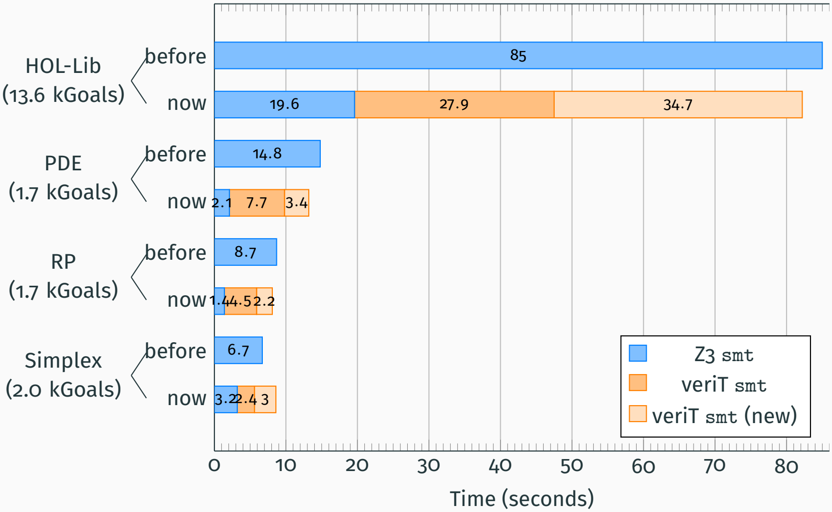
CVC4: Preplay Success Rate



CVC4: Preplay Time (smt only)



CVC4: Preplay Time (smt only)



Step Skipping

Can we do better by understanding proofs globally?

- veriT normalizes every name `x` to `veriT_vr42` with a proof.
  But: $(\forall x.\, P\ x) = (\forall \mathrm{veriT\_vr42}.\, P\ \mathrm{veriT\_vr42})$ for Isabelle De Brujn indices
  So: remove subproof.

Step Skipping

Can we do better by understanding proofs globally?

- veriT normalizes every name `x` to `veriT_vr42` with a proof.
  But: $(\forall x.\, P\, x) = (\forall \mathrm{veriT\_vr42}.\, P\ \mathrm{veriT\_vr42})$ for Isabelle De Brujn indices
  So: remove subproof.

- detect $P \neq Q \vee \neg P \vee Q, \quad P = Q, \quad P$ implies Q. used for every normalization pattern
  So: remove one step and specialize resolution step.
  But: conclusion of step must be known.

Step Skipping

Can we do better by understanding proofs globally?

- veriT normalizes every name `x` to `veriT_vr42` with a proof.
  But: $(\forall x.\, P\, x) = (\forall \mathrm{veriT\_vr42}.\, P\ \mathrm{veriT\_vr42})$ for Isabelle De Brujn indices
  So: remove subproof.

- detect $P \neq Q \vee \neg P \vee Q$, $\quad P = Q$, $\quad P \quad$ implies Q. used for every normalization pattern
  So: remove one step and specialize resolution step.
  But: conclusion of step must be known.

Both important for quantifiers Skolemization: $\geq$ 8 to 3 steps

Different Strategies

- We selected problems for Isabelle
- and run various veriT strategies
- … Several strategies:
  - `del_insts`: instance deletion and breadth-first algorithm to find conflicting instances.
  - `ccfv_SIG` uses a different indexing method for trigger inference + instantiation
  - `ccfv_insts` = `ccfv_SIG` + increased threshold
  - `best`: version that solved most (used by Sledgehammer)

Conclusion

Conclusion

Proof format

- Centralizes manipulation of bound variables and substitutions
- Accommodates many transformations
- Proof checking is (almost) linear

Isabelle/veriT

- working reconstruction in Isabelle (though I got a few bugs on the mailing list)
- it is able to reconstruct proofs

Conclusion

Proof format

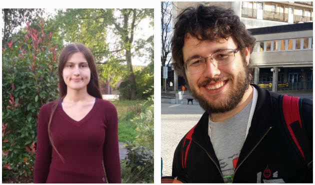
- Centralizes manipulation of bound variables and substitutions
- Accommodates many transformations
- Proof checking is (almost) linear

Isabelle/veriT

- working reconstruction in Isabelle (though I got a few bugs on the mailing list)
- it is able to reconstruct proofs

Questions?

So far: what we want in Isabelle
But what do SMT solvers want?



cvc5 team

Figure 4: Hanna Lachnitt, Haniel Barbose

Based on the 2019 AITP slide by Hans-Jörg Schurr.

RaRe Rules and friends (ongoing work)

- cvc5 distinguishes between
  - simple rules that are read during compilation
  - complicated rules are built-in
- RaRe rules can be translated to Isabelle (where you have to prove it!)

RaRe Rules and friends (ongoing work)

- cvc5 distinguishes between
  - simple rules that are read during compilation

    the fun: some were wrong
  - complicated rules are built-in
- RaRe rules can be translated to Isabelle (where you have to prove it!)
- The hard truth: cvc5 is changing faster than non-developers can keep up with

RaRe Rules and friends (ongoing)

- There is an independent checker Carcara [Andreoti et al, TACAS'23]

- We (finally) are able to check proofs in Isabelle too

  There was some code for Z3, but lost

- One of the most complicated things: n-ary operators

  (+ xs 0 ys) = (+ xs ys)

Extra-theories

```
lemma
  fixes x :: <'a list>
  assumes <map f (map g x) ~= x> and <f o g = id>
  shows False
  using assms
  sledgehammer[dont_compress, verit, debug]
  supply [[smt_trace,verit_compress_proofs]]
  apply (smt (verit) List.map.id[where 'a='a]
     id_apply[of <_ :: 'a list>]
     map_map[of f g <_ :: 'a list>])
  oops
```

Also try it without defining the types!

Native Arithmetics

```
lemma
fixes x :: <nat>
assumes <2+2 + x = 5> <x >= 3>
shows False
supply [[smt_trace,smt_nat_as_int]]
apply (smt (verit))
```

Demo!

Bit-vectors

The sad truth of life:

- usually the semantics do not agree `z3div` outisde SMTLib `8::1 word`, different across solvers
- the logic does not match in subtle ways `extract` is dependent types
- very unstable less than string however
- unclear beforehand how well reconstruction will work eager BV = SAT solving

However:

- understanding the difference is worth it
- even if translation is barely ever useful, maybe it is for someone else?

Master thesis of Torstensson

- Translation from FP in Isabelle to FP in SMTLIB
- Some problems with NaN

- SMT solvers find *more proofs* with the translation
- success rate of reconstruction goes down dramatically (to 5%)

Conclusion

Conclusion

On the Isabelle side:

- the parser is actually bad — worse than usual
- ongoing work for cvc5 — including many useful things for veriT

On the SMT side:

- lots of work
- let’s see which format wins — My current bet: none and no convergence
- But: motivation increased with very complicated theories

  string in cvc5 had a bug, discovered shortly before Amazon would deploy it

References

- Barbosa, Haniel, Jasmin Christian Blanchette, and Pascal Fontaine (2017). "Scalable Fine-Grained Proofs for Formula Processing". In: *CADE*. Vol. 10395. LNCS. Springer, pp. 398–412.
- Barbosa, Haniel et al. (2019). "Scalable Fine-Grained Proofs for Formula Processing". In: *J. Automated Reasoning*.
- Besson, Frédéric, Pascal Fontaine, and Laurent Théry (2011). "A Flexible Proof Format for SMT: a Proposal". In: *PxTP 2011*. Ed. by Pascal Fontaine and Aaron Stump, pp. 15–26.

Déharbe, David, Pascal Fontaine, and Bruno Woltzenlogel Paleo (2011). “Quantifier Inference Rules for SMT proofs”. In: *PxTP 2011*. Ed. by Pascal Fontaine and Aaron Stump, pp. 33–39. URL: https://hal.inria.fr/hal-00642535.

Fleury, Mathias and Hans-Jörg Schurr (2019). “Reconstructing veriT proofs in Isabelle/HOL”. In: EPTCS. PxTP 2019. URL: http://matryoshka.gforge.inria.fr/pubs/PxTP2019.pdf.

The Rules of the Game

The Rules of the Game (1)

$$\frac{}{\triangleright \;\; a \simeq a}\;\textsc{refl}$$

$$\frac{}{\Gamma \;\; \triangleright \;\; b \simeq a}\;\textsc{taut}_{\mathcal{T}}\text{, if } \models_{\mathcal{T}} \Gamma(b) = a$$

The Rules of the Game (1)

$$\frac{}{\rhd \;\; a \simeq a}\;\textsc{refl}$$

$$\frac{}{\Gamma \;\; \rhd \;\; b \simeq a}\;\textsc{taut}_{\mathcal{T}},\text{ if} \vDash_{\mathcal{T}} \Gamma(b) = a$$

The Shortest Proof

$$\frac{\text{axioms}}{\rhd \;\; \bot}\;\textsc{refl}_{\textsc{verit}}$$

Let-Example

$$\dfrac{\dfrac{}{\rhd\ \ a \simeq a}\ \textsf{REFL} \qquad \dfrac{\dfrac{}{x \mapsto a\ \ \rhd\ \ x \simeq a}\ \textsf{REFL} \qquad \dfrac{}{x \mapsto a\ \ \rhd\ \ x \simeq a}\ \textsf{REFL}}{x \mapsto a\ \ \rhd\ \ p(x,x) \simeq p(a,a)}\ \textsf{CONG}}{\rhd\ \ (\mathrm{let}\ x = a\ \mathrm{in}\ p(x,x)) \simeq p(a,a)}\ \textsf{LET}$$

Theory Simplification

$$\dfrac{\dfrac{\dfrac{}{\rhd\ \ k \simeq k}\,\textsc{refl} \qquad \dfrac{}{\rhd\ \ 1\times 0 \simeq 0}\,\textsc{taut}_+}{\rhd\ \ k+1\times 0 \simeq k+0}\,\textsc{cong} \qquad \dfrac{}{\rhd\ \ k+0\simeq k}\,\textsc{taut}_+}{\rhd\ \ k+1\times 0\simeq k}\,\textsc{trans}$$

$$\dfrac{}{\rhd\ \ k \simeq k}\,\textsc{refl}$$

$$\dfrac{\rhd\ \ k+1\times 0 \simeq k \qquad \rhd\ \ k\simeq k}{\rhd\ \ (k+1\times 0 < k)\simeq (k<k)}\,\textsc{cong}$$

The Rules of the Game (2)

$$\frac{\Gamma, y, x \mapsto y \;\; \rhd \;\; \phi \simeq \psi}{\Gamma \;\; \rhd \;\; (Qx.\phi) \simeq (Qy.\psi)} \;\textsc{bind} \text{ if } y \text{ is free}$$

$$\frac{\Gamma, x \mapsto \varepsilon x.\, \neg(Px) \;\; \rhd \;\; \phi \simeq \psi}{\Gamma \;\; \rhd \;\; (\forall x.\phi) \simeq \psi} \;\textsc{sko}_{\forall}$$

Skolemization is allowed in every context including negative (it is sound), although the system is not complete.