Zipperposition, a new platform for Deduction Modulo

Simon Cruanes

Veridis, Inria Nancy
https://cedeela.fr/~simon/

7th of july, 2017

Summary

Quick Tour of Features

```
val set : type -> type.

val i : type.
val a : i.
val b : i.

val[infix "∈"] mem : pi a. a -> set a -> prop.
val[infix "∪"] union : pi a. set a -> set a -> set a.
val[infix "⊆"] subeq : pi a. set a -> set a -> prop.
val[prefix "ℙ"] power : pi a. set a -> set (set a).

rewrite forall a (x:a) A B.  mem x (union A B) <=> (mem x A || mem x B).

rewrite forall a A B.  subeq A B <=> (forall (x:a). mem x A => mem x B).

rewrite forall a (x:set a) A.  mem x (power A) <=> subeq x A.

goal forall (A:set i) B.  subeq (power A) (power (union A B)).
```

Solution

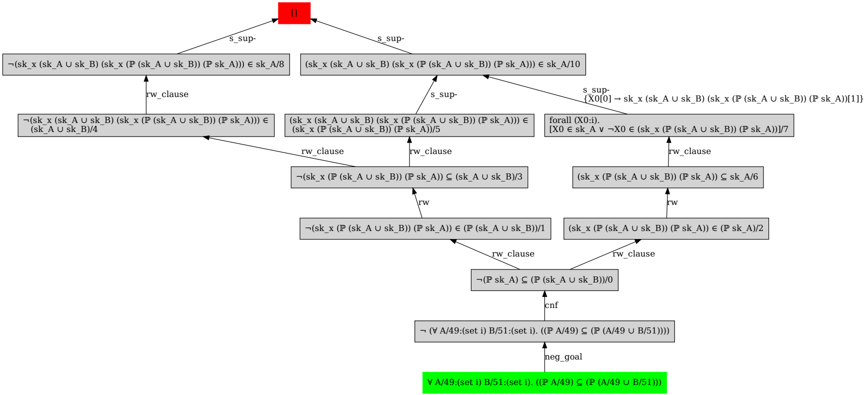
$ zipperposition --dot foo.dot set_fancy.dot

$ dot -Txlib foo.dot

Solution

$ zipperposition −−dot foo.dot set_fancy.dot

$ dot −Txlib foo.dot



Input Language

Custom language to support custom features.

- rank-1 polymorphic types
- toplevel statements (declare everything)
  - assertions
  - rewrite rules
  - definitions
  - datatypes
  - goal (negated assertion)
  - lemmas (introduces a cut)
- ML-like syntax for terms
  - curried terms
  - if/then/else, match
  - usual operators
- custom attributes (AC, infix-notation, . . .)

Zipperposition: the prover

- written in OCaml ($\sim$) from scratch
- 37k loc right now
- BSD license, on github
  https://github.com/c-cube/zipperposition
- decently modular, decent performances
- paper about the internals: https://hal.inria.fr/hal-01101057/

Global Framework : Superposition

Zipperposition is centered around Superposition.

the calculus:

- clausal (works on disjunctions of literals)
- refutational (goal: deduce $\bot$)
- equational (tailored for reasoning with equality)

Global Framework : Superposition

Zipperposition is centered around Superposition.

the calculus:

- clausal (works on disjunctions of literals)
- refutational (goal: deduce $\bot$)
- equational (tailored for reasoning with equality)

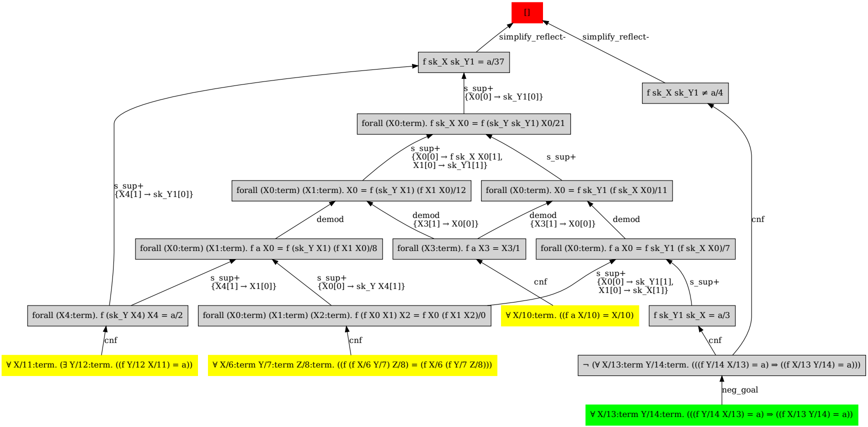
Say we have only two elements a and b, on which p holds. Then prove $\forall x. p(x)$ by refuting $\exists c. \neg p(c)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\neg p(c) \qquad x \simeq a \lor x \simeq b}{\neg p(a) \lor c \simeq b} \qquad p(a)}{c \simeq b} \qquad \neg p(c)}{\neg p(b)} \qquad p(b)}{\bot}$$

(Note the *binding* of x to c using *unification*)

Example: Group Theory

Left-inverse is also right-inverse:



Some Notable Extensions

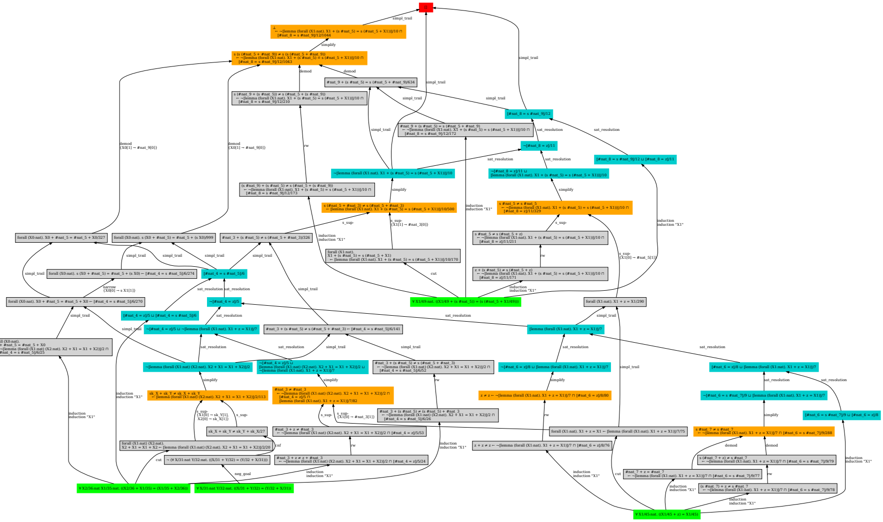
Zipperposition also has some extensions:

- AC symbols
- Linear {Integer, Rational} Arithmetic
- Structural Induction (for datatypes)
- Higher-Order Logic (**WIP!**)

$\rightarrow$ quite easy to plug in new simplification/inference rules

Inductive Proof

Commutativity of addition:



Summary

Deduction Modulo (cheatsheet)

Recall

- rewrite rules for terms
- rewrite rules for *literals* (signed atoms)
- also perform *narrowing* (unification replacing matching)
- also do *narrowing* inside rules' LHS (contextual narrowing)
- great for some theories!

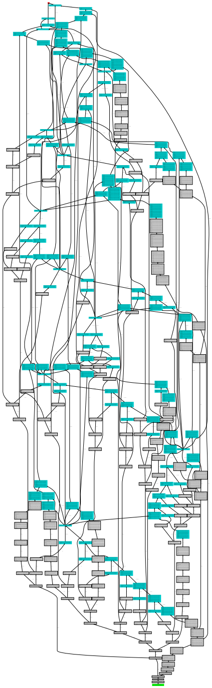
$\rightarrow$ Let's look at some examples.

Favorite Example : Set Theory

```
val set : type -> type.

val[infix "∈"] mem : pi a. a -> set a -> prop.
val[infix "∪"] union : pi a. set a -> set a -> set a.
val[infix "⊆"] subeq : pi a. set a -> set a -> prop.

rewrite forall a s1 s2 x.  mem a x (union a s1 s2) <=> mem a x s1 || mem a x
s2.

rewrite forall a s1 s2.  subeq a s1 s2 <=> (forall x. mem a x s1 => mem a x
s2).

rewrite forall a (s1 s2 : set a).  s1 = s2 <=> (subeq s1 s2 && subeq s2 s1).

goal
  forall a (S1 S2 S3 S4 S5 S6 : set a).
    (union S1 (union S2 (union S3 (union S4 (union S5 S6))))) =
    (union S6 (union S5 (union S4 (union S3 (union S2 S1))))).
```

- solved in 0 steps
- entirely reduced to $\in$-literals
- AVATAR does the splitting

$\rightarrow$ bit-blasting for free!



Lightweight Theories

Example

Classic theory of (extensional) arrays

```
val array : type -> type -> type.
val update : pi a b. array a b -> a -> b -> array a b.
val get : pi a b. array a b -> a -> b.

rewrite forall a b (arr:array a b) x1 x2 v.
  get (update arr x2 v) x1 = (if x1=x2 then v else get arr x1).

rewrite forall a b (arr1 arr2 : array a b).
  arr1 = arr2 <=> (forall x. get arr1 x = get arr2 x).
```

Lightweight Theories

Example

Classic theory of (extensional) arrays

```
val array : type -> type -> type.
val update : pi a b. array a b -> a -> b -> array a b.
val get : pi a b. array a b -> a -> b.

rewrite forall a b (arr:array a b) x1 x2 v.
  get (update arr x2 v) x1 = (if x1=x2 then v else get arr x1).

rewrite forall a b (arr1 arr2 : array a b).
  arr1 = arr2 <=> (forall x. get arr1 x = get arr2 x).
```

```
goal forall x arr. arr = update arr x (get arr x).
```

Lightweight Theories

Example

Classic theory of (extensional) arrays

```
val array : type -> type -> type.
val update : pi a b. array a b -> a -> b -> array a b.
val get : pi a b. array a b -> a -> b.

rewrite forall a b (arr:array a b) x1 x2 v.
  get (update arr x2 v) x1 = (if x1=x2 then v else get arr x1).

rewrite forall a b (arr1 arr2 : array a b).
  arr1 = arr2 <=> (forall x. get arr1 x = get arr2 x).
```

```
goal forall x arr. arr = update arr x (get arr x).
```

```
goal forall x1 x2 arr. x1 != x2 && v1 != v2 =>
    update (update arr x1 v1) x2 v2 != update (update arr x2 v1) x1 v2.
```

B-ware, again

- Internship of Pierre-Louis Euvrard, in Montpellier
- co-supervised with David Delahaye
- experiment with (typed) set theory using Zipperposition
- Lemmas: good results
- Proof Obligations: WIP

Summary

Custom Notion of Equality

In previous examples, there were rules such as:

```
rewrite forall a (s1 s2 : set a).
  s1 = s2 <=> (subset s1 s2 && subset s2 s1).

rewrite forall a b (arr1 arr2 : array a b).
  arr1 = arr2 <=> (forall x. get arr1 x = get arr2 x).
```

Custom Notion of Equality

In previous examples, there were rules such as:

```
rewrite forall a (s1 s2 : set a).
  s1 = s2 <=> (subset s1 s2 && subset s2 s1).

rewrite forall a b (arr1 arr2 : array a b).
  arr1 = arr2 <=> (forall x. get arr1 x = get arr2 x).
```

$\rightarrow$ Custom equality!

- we only rewrite **negative** equational literals
- $a \simeq b$ is already maximal information
- $a \not\simeq b$ is a *goal* (prove $a \simeq b$ to remove the literal)
- should only be useful for LHS-pattern $x \simeq y$ of certain types

Where does it lead?

question: Is this studied?

Where does it lead?

question: Is this studied?

Follow-up: Unification with Constraints

- used for arithmetic already
- principle: during unification, *delay* pairs of certain types
- to unify $f(a, b+1)$ and $f(a, 1+b)$, delay $b+1 = 1+b$

→ We add a literal $b+1 \not\approx 1+b$ to resulting clause

→ This literal will be deal with by rewriting/theories

→ would be interesting to delay pairs of types that have equational rewrite rules (e.g. sets, arrays)

Conclusion

Current status

- usable prover for Superposition modulo
- no completeness result (except for resolution modulo?)
- full narrowing implemented (and sometimes useful)
- nice proof output for debugging

$\rightarrow$ good platform for experimenting with ATP modulo

Conclusion

Current status

- usable prover for Superposition modulo
- no completeness result (except for resolution modulo?)
- full narrowing implemented (and sometimes useful)
- nice proof output for debugging

$\rightarrow$ good platform for experimenting with ATP modulo

Future work/directions

- custom induction schemas
- delayed unification for extensional types
- WIP: higher-order (in particular, rewriting/reasoning with *patterns*)
- direly needed: proof checking

Thanks for your attention!