Theory Exploration and Inductive Theorem Proving

Licentiate Seminar

Dan Rosén

March 8, 2016
Example

\[ \forall \; xs \; ys \cdot \text{len} (xs \; ++ \; ys) = \text{len} \; xs + \text{len} \; ys \]
Example

\[ \forall \, xs \, ys \cdot \text{len} \,(xs \, ++ \, ys) = \text{len} \, xs \, + \, \text{len} \, ys \]
Example

∀ xs ys · len (xs ++ ys) = len xs + len ys

Coding

Interactive
Theorem
Proving
Example

\[ \forall \ x\!\!\:s\ y\!\!\:s \ \cdot \ \text{len} \ (xs \!\!\:+\!\!\: ys) = \text{len} \ xs + \text{len} \ ys \]
HipSpec architecture

Haskell Source
HipSpec architecture

Haskell Source

Translation (Hip)

First-Order Theory

Theorem Prover

Induction (Hip)

Theorem Timeout

Conjectures QuickSpec

Extend theory Open conjecture
HipSpec architecture

Haskell Source

Translation (Hip)

First-Order Theory

Induction (Hip)

Theorem Prover

Extend theory

Open conjecture

Timeout

Conjectures

QuickSpec

Theorem Prover

Induction (Hip)
HipSpec architecture

Haskell Source → Translation (Hip) → First-Order Theory → Induction (Hip) → Theorem Prover

Extend theory
Open conjecture
QuickSpec
Timeout
Conjectures
Theorem Prover
Induction (Hip)
HipSpec architecture

- Haskell Source
- Translation (Hip)
- First-Order Theory
- Induction (Hip)
- Theorem Prover
- Timeout
- Conjectures
- QuickSpec
- Open conjecture
- Extend theory
HipSpec architecture

Haskell Source

Translation (Hip)

First-Order Theory

Induction (Hip)

Conjectures

Theorem Prover

Timeout

Extend theory

Open conjecture

QuickSpec
HipSpec architecture

Haskell Source

Translation (Hip)

First-Order Theory

Induction (Hip)

Conjectures

QuickSpec

Theorem Prover

Timeout

Extend theory

Open conjecture
HipSpec architecture

Haskell Source → Translation (Hip) → First-Order Theory → Induction (Hip) → Conjectures → QuickSpec

First-Order Theory → Extend theory → Theorem Prover → Theorem Timeout

Conjectures
HipSpec architecture

Haskell Source

Translation (Hip)

First-Order Theory

Extend theory

Induction (Hip)

Conjectures

Open conjecture

QuickSpec

Theorem Prover

Timeout

Theorem
Generates well-typed terms up to some depth:

\[
\begin{align*}
\text{len} \ (xs++xs) & \quad \text{len} \ (\text{rev} \ ys) & \quad \text{len} \ xs \\
\text{rev} \ (xs++xs) & \quad []++xs & \quad \text{rev} \ (xs++ys) \\
\text{rev} \ xs++\text{rev} \ ys & \quad \text{rev} \ xs & \quad \text{len} \ (\text{rev} \ xs) \\
(xs++ys)++[] & \quad xs & \quad xs++(ys++ys) \\
xs++[] & \quad \text{len} \ xs+\text{len} \ ys & \quad \text{len} \ (ys++xs) \\
\text{len} \ xs+\text{len} \ xs & \quad (xs++ys)++ys & \quad \text{rev} \ xs++\text{rev} \ xs \\
\text{rev} \ (ys++xs) & \quad \text{rev} \ (\text{rev} \ xs) & \quad xs++ys
\end{align*}
\]
Theory Exploration: QuickSpec

- len(xs++xs)
- len(xs++rev xs)
- len xs+len xs

- xs
- xs++[]
- rev(rev xs)

- xs++(ys++ys)
- (xs++ys++)++ys

- len xs
- len (rev xs)

- len (xs++ys)
- len (ys++xs)
HipSpec architecture

Haskell Source

Translation (Hip)

First-Order Theory

Extend theory

Induction (Hip)

Theorem Prover

Timeout

Conjectures

QuickSpec

Open conjecture
Prioritising Equations

1. Call graph

- len
- ++
- +
- rev
Prioritising Equations

1. Call graph

\[
\begin{align*}
\text{xs}++[] &= \text{xs} \\
\text{rev} (\text{xs}++\text{ys}) &= \text{rev} \ \text{ys} \ ++ \ \text{rev} \ \text{xs} \\
\text{len} (\text{xs}++\text{ys}) &= \text{len} \ \text{xs} + \ \text{len} \ \text{ys}
\end{align*}
\]
Prioritising Equations

1. Call graph

2. Size

\[
\begin{align*}
x s++[] &= x s \\
(xs++ys)++[] &= xs++ys \\
(xs++ys)++zs &= xs++(ys++zs) \\
(xs++ys)++(zs++ws) &= xs++(ys++(zs++ws))
\end{align*}
\]
Prioritising Equations

1. Call graph

2. Size

3. Number of variables

\[(xs++ys)++zs = xs++(ys++zs)\]
\[(xs++xs)++ys = xs++(xs++ys)\]
\[(xs++xs)++xs = xs++(xs++xs)\]
Related work: syntactic approaches

\[ \forall \, xs \cdot \text{rev} \left( \text{rev} \, xs \right) = xs \]
Related work: syntactic approaches

\[ \forall xs \cdot \text{rev} (\text{rev} \ xs) = xs \]

IH : \text{rev} (\text{rev} \ as) = as

\text{rev} (\text{rev} \ (a : as)) = a : as
Related work: syntactic approaches

\[ \forall \, xs \cdot \text{rev} \left( \text{rev} \, xs \right) = xs \]

IH : \text{rev} \left( \text{rev} \, as \right) = as

\text{rev} \left( \text{rev} \left( a : as \right) \right) = a : as
Related work: syntactic approaches

∀ xs · rev (rev xs) = xs

IH : rev (rev as) = as

rev (rev (a : as)) = a : as

⇐⇒ rev (rev as ++ [a]) = a : as

Lemma speculation

▶ CLAM (1996)
▶ ACL2
▶ IsaPlanner
▶ Zeno (2012)
Related work: syntactic approaches

\[ \forall xs \cdot \text{rev} (\text{rev} \, xs) = xs \]

IH : \text{rev} (\text{rev} \, as) = as

\text{rev} (\text{rev} \, (a : as)) = a : as

\iff\iff \text{rev} (\text{rev} \, as ++ [a]) = a : as
Related work: syntactic approaches

∀ xs · rev (rev xs) = xs

IH : rev (rev as) = as
rev (rev (a : as)) = a : as
⇔ rev (rev as ++ [a]) = a : as
⇔ rev (rev as ++ [a]) = a : rev (rev as)
Related work: syntactic approaches

\[ \forall \text{xs} \cdot \text{rev} (\text{rev} \text{xs}) = \text{xs} \]

IH : \text{rev} (\text{rev as}) = \text{as}

\text{rev} (\text{rev} (\text{a : as})) = \text{a : as}

\iff \text{rev} (\text{rev as} \mathbin{++} \text{[a]}) = \text{a : as}

\iff \text{rev} (\text{rev as} \mathbin{++} \text{[a]}) = \text{a : rev} (\text{rev as})
Related work: syntactic approaches

∀ xs · rev (rev xs) = xs

IH : rev (rev as) = as

rev (rev (a : as)) = a : as

rev (rev as ++ [a]) = a : as

rev (rev as ++ [a]) = a : rev (rev as)

∀ ys · rev (ys ++ [a]) = a : rev ys
Related work: syntactic approaches

∀ xs · rev (rev xs) = xs

IH : rev (rev as) = as
rev (rev (a : as)) = a : as

⇔ rev (rev as ++ [a]) = a : as
⇔ rev (rev as ++ [a]) = a : rev (rev as)
⇔ ∀ ys · rev (ys ++ [a]) = a : rev ys

Lemma speculation (an example of a proof critic)

- CLAM (1996)
- ACL2
- IsaPlanner
- Zeno (2012)
Related work: syntactic approaches

\[
\begin{align*}
qrev \; [ ] \quad & ys = ys \\
qrev \; (x : xs) \; ys = qrev \; xs \; (x : ys) \\
\forall \; xs \cdot qrev \; xs \; [ ] = rev \; xs
\end{align*}
\]
Related work: syntactic approaches

\[ qrev \, [\,] \, ys = ys \]
\[ qrev \, (x : xs) \, ys = qrev \, xs \, (x : ys) \]
\[ \forall \, xs \, \cdot \, qrev \, xs \, [\,] = rev \, xs \]
\[ qrev \, (a : as) \, [\,] \]
Related work: syntactic approaches

qrev [ ] ys = ys
qrev (x : xs) ys = qrev xs (x : ys)
∀ xs · qrev xs [ ] = rev xs
    qrev (a : as) []

= rev (a : as)
Related work: syntactic approaches

\[
qrev \emptyset \ ys = ys \\
qrev (x : xs) \ ys = qrev xs (x : ys) \\
\forall xs \cdot qrev xs \emptyset = rev xs \\
\quad qrev (a : as) \emptyset = qrev as (a : \emptyset) \\
\Rightarrow rev (a : as)
\]
Related work: syntactic approaches

\[ qrev \; \emptyset \; ys = ys \]
\[ qrev \; (x : xs) \; ys = qrev \; xs \; (x : ys) \]
\[ \forall \; xs \cdot qrev \; xs \; \emptyset = rev \; xs \]
\[ qrev \; (a : as) \; \emptyset = qrev \; as \; (a : \emptyset) \]
\[ = rev \; as \; \leftarrow [a] \]
\[ = rev \; (a : as) \]
Related work: syntactic approaches

\[ qrev \; [] \; ys = ys \]
\[ qrev \; (x : xs) \; ys = qrev \; xs \; (x : ys) \]
\[ \forall \; xs \cdot qrev \; xs \; [] = rev \; xs \]
\[ qrev \; (a : as) \; [] = qrev \; as \; (a : []) \]
\[ = \ldots \; ??? \; \ldots \]
\[ = \text{rev} \; as \; ++ \; [a] \]
\[ = \text{rev} \; (a : as) \]
Related work: syntactic approaches

\[
\begin{align*}
qrev \; [] \quad & ys = ys \\
qrev \; (x : xs) \; ys = qrev \; xs \; (x : ys) \\
\forall \; xs \cdot qrev \; xs \; [] = rev \; xs \\
qrev \; (a : as) \; [] = qrev \; as \; (a : [\;]) \\
= \ldots \text{???} \ldots & \text{IH : } rev \; as = qrev \; as \; [] \\
= rev \; as \; \# [\;a] \\
= rev \; (a : as)
\end{align*}
\]
Related work: syntactic approaches

\[
\begin{align*}
q\text{rev} \, [] & \quad ys = ys \\
q\text{rev} \, (x : xs) \, ys & = q\text{rev} \, xs \, (x : ys) \\
\forall \, xs \cdot q\text{rev} \, xs \, [] & = \text{rev} \, xs \\
q\text{rev} \, (a : as) \, [] & \\
& = q\text{rev} \, as \, (a : []) \\
& = \ldots \text{???} \ldots \\
& \quad \text{IH}: \text{rev} \, as = q\text{rev} \, as \, [] \\
& = \text{rev} \, as \, \uplus \, [a] \\
& = \text{rev} \, (a : as)
\end{align*}
\]

**CLAM**: Generalisation critic solves this
Related work: syntactic approaches

\[
\begin{align*}
qrev \, [] \quad & ys = ys \\
qrev \, (x : xs) \, ys = qrev \, xs \, (x : ys) \\
\forall \, xs \cdot qrev \, xs \, [] = rev \, xs \\
qrev \, (a : as) \, [] = qrev \, as \, (a : []) \\
= ... ??? ... \quad & \text{IH: } rev \, as = qrev \, as \, [] \\
= \, rev \, as \, ++ \, [a] \\
= \, rev \, (a : as)
\end{align*}
\]

**CLAM:** Generalisation critic solves this, given ++ assoc
Theory exploration

- QuickSpec (2010)
- IsaCoSy (2011)
- CVC4 (2015)
Theory exploration

- QuickSpec (2010)
- IsaCoSy (2011)
  
  *can do rev-qrev... but it takes hours*

- CVC4 (2015)
Theory exploration

- QuickSpec (2010)
- IsaCoSy (2011)
  - can do rev-qrev... but it takes hours
- CVC4 (2015)
  - quantifier instantiation module extended with structural induction
∀ n xs · take (len xs − n) (rev xs) = rev (drop n xs)
∀ n xs · take (len xs − n) (rev xs) = rev (drop n xs)

rev (drop 6 "lena kanel")
rev "anel"
"lena"
∀ n xs · take (len xs − n) (rev xs) = rev (drop n xs)

rev (drop 6 "lena kanel")
rev "anel"
"lena"

take (len "lena kanel" − 6) (rev "lena kanel")
take (len "lena kanel" − 6) ("lenak anel")
take (10 − 6) ("lenak anel")
take 4 ("lenak anel")
"lena"
take-drop-len-rev-minus

take (len xs — n)  (rev xs)
take-drop-len-rev-minus

take \(\text{len } xs - n\) (\(\text{rev } xs\))
take-drop-len-rev-minus

\[
\text{take} \ (\text{len} \ xs - n) \quad (\text{rev} \ xs) \\
= \text{take} \ (\text{len} \ (\text{drop} \ n \ xs)) \quad (\text{rev} \ xs)
\]
take-drop-len-rev-minus

take (len xs \(-\ n) \quad (rev \ xs)
\equiv \quad take \ (len \ (\text{drop} \ n \ xs)) \ (rev \ xs)
take-drop-len-rev-minus

\[
\text{take} \ (\text{len} \ \text{xs} - n) \quad \text{(rev} \ \text{xs)}
\]
\[
= \text{take} \ (\text{len} \ \text{(drop} \ n \ \text{xs})) \ (\text{rev} \ \text{xs})
\]
\[
= \text{take} \ (\text{len} \ \text{(drop} \ n \ \text{xs})) \ (\text{rev} \ \text{(take} \ n \ \text{xs} \ \text{++} \ \text{drop} \ n \ \text{xs}))
\]
take-drop-len-rev-minus

take (len xs − n) (rev xs)
= take (len (drop n xs)) (rev xs)
= take (len (drop n xs)) (rev (take n xs ++ drop n xs))
take-drop-len-rev-minus

take \( (\text{len } xs - n) \) (rev xs)
= take \( (\text{len } (\text{drop } n \ xs)) \) (rev xs)
= take \( (\text{len } (\text{drop } n \ xs)) \) (rev (take n xs ++ drop n xs))
= take \( (\text{len } (\text{drop } n \ xs)) \) (rev (drop n xs) ++ rev (take n xs))
take-drop-len-rev-minus

take (\text{len} \, \text{xs} - n) \quad (\text{rev} \, \text{xs})

\begin{align*}
\equiv \ & \text{take} \ (\text{len} \ (\text{drop} \ n \ \text{xs})) \ (\text{rev} \ \text{xs}) \\
\equiv \ & \text{take} \ (\text{len} \ (\text{drop} \ n \ \text{xs})) \ (\text{rev} \ (\text{take} \ n \ \text{xs} + \ \text{drop} \ n \ \text{xs})) \\
\equiv \ & \text{take} \ (\text{len} \ (\text{drop} \ n \ \text{xs})) \ (\text{rev} \ (\text{drop} \ n \ \text{xs}) + \ \text{rev} \ (\text{take} \ n \ \text{xs}))
\end{align*}
take-drop-len-rev-minus

take \((\text{len } xs - n)\) \((\text{rev } xs)\)

\[= \text{take} \left( \text{len} \left( \text{drop } n \ xs \right) \right) \left( \text{rev } xs \right) \]

\[= \text{take} \left( \text{len} \left( \text{drop } n \ xs \right) \right) \left( \text{rev} \left( \text{take } n \ xs \quad \text{++} \quad \text{drop } n \ xs \right) \right) \]

\[= \text{take} \left( \text{len} \left( \text{drop } n \ xs \right) \right) \left( \text{rev} \left( \text{drop } n \ xs \right) \quad \text{++} \quad \text{rev} \left( \text{take } n \ xs \right) \right) \]

\[= \text{take} \left( \text{len} \left( \text{rev} \left( \text{drop } n \ xs \right) \right) \right) \left( \text{rev} \left( \text{drop } n \ xs \right) \quad \text{++} \quad \text{rev} \left( \text{take } n \ xs \right) \right) \]

\[= \text{rev} \left( \text{drop } n \ xs \right) \]
take-drop-len-rev-minus

\[
\text{take } (\text{len } \text{xs} - n) \quad (\text{rev } \text{xs}) \\
\equiv \text{take } (\text{len } (\text{drop } n \text{ xs})) (\text{rev } \text{xs}) \\
\equiv \text{take } (\text{len } (\text{drop } n \text{ xs})) (\text{rev } (\text{take } n \text{ xs} + + \text{drop } n \text{ xs})) \\
\equiv \text{take } (\text{len } (\text{drop } n \text{ xs})) (\text{rev } (\text{drop } n \text{ xs} ++ \text{rev } (\text{take } n \text{ xs}))) \\
\equiv \text{take } (\text{len } (\text{rev } (\text{drop } n \text{ xs}))) \\
\quad (\text{rev } (\text{drop } n \text{ xs}) + + \text{rev } (\text{take } n \text{ xs}))
\]
take-drop-len-rev-minus

take \( (\text{len } xs - n) \) \hspace{1cm} (rev \; xs) \\
= take \( (\text{len } (\text{drop } n \; xs)) \) \hspace{1cm} (rev \; xs) \\
= take \( (\text{len } (\text{drop } n \; xs)) \) \hspace{1cm} (rev \; (\text{take } n \; xs \; ++ \; \text{drop } n \; xs)) \\
= take \( (\text{len } (\text{drop } n \; xs)) \) \hspace{1cm} (rev \; (\text{drop } n \; xs) \; ++ \; \text{rev } (\text{take } n \; xs)) \\
= take \( (\text{len } (\text{rev } (\text{drop } n \; xs))) \) \hspace{1cm} (\text{rev } (\text{drop } n \; xs) \; ++ \; \text{rev } (\text{take } n \; xs)) \\
= \text{rev } (\text{drop } n \; xs)
take-drop-len-rev-minus

2 : \text{take } n \ (\text{take } n \ xs) = \text{take } n \ xs  \\
20 : xs ++ [] = xs  \\
23 : (xs ++ ys) ++ zs = xs ++ (ys ++ zs) \quad \text{using } 20  \\
26 : \text{rev } ys ++ \text{rev } xs = \text{rev } (xs ++ ys) \quad \text{using } 20, 23  \\
29 : \text{take } (\text{len } xs) \ xs = xs \quad \text{using } 2  \\
30 : \text{len } (\text{drop } n \ xs) = \text{len } xs - n  \\
31 : \text{len } (ys ++ xs) = \text{len } (xs ++ ys) \quad \text{using } 20  \\
37 : \text{take } (\text{len } ys) \ (ys ++ xs) = ys \quad \text{using } 20, 29  \\
40 : \text{take } n \ xs ++ \text{drop } n \ xs = xs \quad \text{using } 20  \\
48 : \text{len } (\text{rev } xs) = \text{len } xs \quad \text{using } 31  \\
50 : \text{take } (\text{len } xs - n) \ (\text{rev } xs) = \text{rev } (\text{drop } n \ xs) \quad \text{using } 26, 30, 37, 40, 48
Related work:
schematic theory exploration

\( \forall \, x \, y \, z \cdot x \ast (y + z) = (x \ast y) + (x \ast z) \)
Related work: schematic theory exploration

\[ \forall x \ y \ z \cdot x \ast (y + z) = (x \ast y) + (x \ast z) \]

\[ \forall x \ y \ z \cdot x + (y \ast z) = (x + y) \ast (x + z) \]
Related work: schematic theory exploration

\[ \forall x \, y \, z \cdot x \ast (y + z) = (x \ast y) + (x \ast z) \]
\[ \forall x \, y \, z \cdot x + (y \ast z) = (x + y) \ast (x + z) \]

- IsaScheme (2012)
- Pirate (unpublished)
Benchmarks

- 86 from IsaPlanner (2010)
- 50 from CLAM (1996)

+ * half double fac exp
++ rev qrev len zip
delete drop take elem count
map filter takeWhile dropWhile
insert isort
union intersect subset
height mirror
Benchmarks

- 86 from IsaPlanner (2010)
- 50 from CLAM (1996)
  no definitions

+ * half double fac exp
++ rev qrev len zip
delete drop take elem count
map filter takeWhile dropWhile
insert isort
union intersect subset
height mirror
Benchmarks

- 86 from IsaPlanner (2010)
  most are way too easy
- 50 from CLAM (1996)
  no definitions

+ * half double fac exp
++ rev qrev len zip
delete drop take elem count
map filter takeWhile dropWhile
insert isort
union intersect subset
height mirror
Benchmarks

- 86 from IsaPlanner (2010)
  most are way too easy
- 50 from CLAM (1996)
  no definitions
  these days, also too easy

+ * half double fac exp
++ rev qrev len zip
delete drop take elem count
map filter takeWhile dropWhile
insert isort
union intersect subset
height mirror
Quest: make a good benchmark suite

- Actual benchmarks
- An input format
TIP: Tons of Inductive Problems
TIP: Tons of Inductive Problems

- sorting algorithms: bitonic bubble heap merge quick selection stooge tree
- unambiguity of grammars
- propositional solver
- lambda calculus substitution
- regular expressions
- integers as datatypes
- binary naturals
- non-structurally recursive functions
What criteria would you pick for a benchmark format?
TIP format

What criteria would you pick for a benchmark format?

- Natural, no encodings
- Extending existing format
- Simple for developers
What criteria would you pick for a benchmark format?

- Natural, no encodings
- Extending existing format
- Simple for developers

Haskell
Isabelle
TPTP
Why3
SMT-LIB
TIP format

What criteria would you pick for a benchmark format?

- Natural, no encodings
- Extending existing format
- Simple for developers

- Haskell
- Isabelle
- TPTP
- Why3
- SMT-LIB
What criteria would you pick for a benchmark format?

- Natural, no encodings
- Extending existing format
- Simple for developers

Haskell, Isabelle, TPTP, Why3, SMT-LIB
What criteria would you pick for a benchmark format?

- Natural, no encodings
- Extending existing format
- Simple for developers

- Haskell
- Isabelle
- TPTP
- Why3
- SMT-LIB
TIP format

What criteria would you pick for a benchmark format?

- Natural, no encodings
- Extending existing format
- Simple for developers

- Haskell
- Isabelle
- TPTP
- Why3
- SMT-LIB
What criteria would you pick for a benchmark format?

- Natural, no encodings
- Extending existing format
- Simple for developers

Haskell
Isabelle
TPTP
Why3
SMT-LIB
Future work: Conditionals

∀ xs · sorted (isort xs)
Future work: Conditionals

\[ \forall \, xs \cdot \text{sorted} \left( \text{isort} \, xs \right) \]

\[ \forall \, xs \cdot \text{sorted} \, xs \iff \text{sorted} \left( \text{insert} \, x \, xs \right) \]
Future work: Conditionals

\[ \forall xs \cdot \text{sorted } (\text{isort } xs) \]
\[ \forall xs \cdot \text{sorted } xs \rightleftharpoons \text{sorted } (\text{insert } x \ xs) \]
\[ \forall xs \cdot \text{sorted } xs = \text{sorted } (\text{insert } x \ xs) \]
Future work: Conditionals

∀ s x y · ordered s ⇒ insert x (insert y s) =
insert y (insert x s)
∀ s x y · ordered s ⇒ insert x (insert y s) =
imsetEq insert y (insert x s)

∀ s x y · ordered s ⇒ insert x (insert y s) ‘setEq‘
insert y (insert x s)
Future work: Conditionals

\[ \forall s \, x \, y \cdot \text{ordered } s \implies \text{insert } x \left( \text{insert } y \ s \right) = \text{insert } y \left( \text{insert } x \ s \right) \]

\[ \forall s \, x \, y \cdot \text{ordered } s \implies \text{insert } x \left( \text{insert } y \ s \right) \text{‘setEq‘} \text{insert } y \left( \text{insert } x \ s \right) \]

\[ \forall a \, b \, c \cdot a \text{‘setEq‘} b \implies b \text{‘setEq‘} c \implies a \text{‘setEq‘} c \]
Future work: Conditionals

∀ t u xs ys ·
  show t ++ xs = show u ++ ys ⇔ t = u ∧ xs = ys

∀ xs ys · len xs = len ys ⇒
  zip xs ys ++ zip as bs = zip (xs ++ as) (ys ++ bs)
Future work: Stronger induction

\[
\begin{align*}
\text{selsort} \; [] &= [] \\
\text{selsort} \; xs &= \\
& \quad \text{minimum} \; xs : \text{selsort} \; (\text{delete} \; (\text{minimum} \; xs) \; \text{xs})
\end{align*}
\]
Future work: Stronger induction

\[
\begin{align*}
s\text{selsort} \; [ ] & = [ ] \\
s\text{selsort} \; xs & = \\
& \text{minimum} \; xs \; : \; s\text{selsort} \; (\text{delete} \; (\text{minimum} \; xs) \; xs)
\end{align*}
\]

\[
\begin{align*}
P([]) \\
\forall \; y \; ys \cdot \; \text{let} \; \; xs = y : ys \\
\quad \text{in} \; \; P(\text{delete} \; (\text{minimum} \; xs) \; xs) \implies P(xs) \\
\forall \; zs \cdot \; P(zs)
\end{align*}
\]
Future work: Stronger induction

\[
\begin{align*}
\text{selsort } \[] & = [] \\
\text{selsort } xs & = \\
& \text{minimum } xs : \text{selsort } (\text{delete } (\text{minimum } xs) \; xs)
\end{align*}
\]

\[
\begin{align*}
P([]) & \\
\forall \; y \; ys \cdot \; \text{let } \; xs = y : ys \\
& \quad \text{in } \; P(\text{delete } (\text{minimum } xs) \; xs) \implies P(xs)
\end{align*}
\]

\[
\begin{array}{c}
\forall \; zs \cdot \; P(zs)
\end{array}
\]

\[
\begin{align*}
\forall \; xs \cdot (\forall \; ys \cdot \text{length } ys < \text{length } xs \implies P(ys)) & \implies P(xs) \\
& \quad \text{in } \; P(zs)
\end{align*}
\]
Future work: Function synthesis

rotate :: Nat → [a] → [a]
rotate Zero xs = xs
rotate (Succ n) [] = []
rotate (Succ n) (x : xs) = rotate n (xs ++ [x])

rotate 4 "lena kanel" = " kanellena"
Future work: Function synthesis

\[
\text{rotate} :: \text{Nat} \rightarrow [a] \rightarrow [a]
\]
\[
\text{rotate} \; \text{Zero} \; \text{xs} \quad = \quad \text{xs}
\]
\[
\text{rotate} \; (\text{Succ} \; n) \; [] \quad = \quad []
\]
\[
\text{rotate} \; (\text{Succ} \; n) \; (x:xs) \quad = \quad \text{rotate} \; n \; (xs \; \text{++} \; [x])
\]
\[
\text{rotate} \; 4 \; "\text{lena} \; \text{kanel}" \quad = \quad "\text{kanellena}"
\]
\[
\forall \; xs \quad \cdot \quad \text{rotate} \; (\text{length} \; xs) \; xs \quad = \quad xs
\]
Future work: Function synthesis

rotate :: Nat → [a] → [a]
rotate Zero xs = xs
rotate (Succ n) [] = []
rotate (Succ n) (x : xs) = rotate n (xs ++ [x])

rotate 4 "lena kanel" = " kanellena"

∀ xs · rotate (length xs) xs = xs
∀ xs ys · rotate (length xs) (xs ++ ys) = ys ++ xs
Future work: Function synthesis

rotate :: Nat → [a] → [a]
rotate Zero xs = xs
rotate (Succ n) [] = []
-- rotate (Succ n) (x : xs) = rotate n (xs ++ [x])
rotate (Succ n) (x : xs) = rotate n (xs 'snoc' x)

rotate 4 "lena kanel" = " kanellena"

∀ xs . rotate (length xs) xs = xs
∀ xs ys . rotate (length xs) (xs ++ ys) = ys ++ xs
Future work

- Conditionals
- Stronger induction
- Function synthesis
- Proof output
Future work

- Conditionals
- Stronger induction
- Function synthesis
- Proof output
Vision

Simple functions
Vision

- Simple functions
- Data structures
- Compiler passes
- ...

(Coding, Interactive, Theorem Proving, HipSpec, Hipster, TIP)
Vision

Simple functions

Data structures

Compiler passes

...

Coding

Interactive Theorem Proving
Vision

Simple functions

Data structures

Compiler passes

...

HipSpec

Coding

Interactive Theorem Proving
Vision

Simple functions
Data structures
Compiler passes
...

HipSpec
Coding

Hipster
Interactive Theorem Proving
Vision

Simple functions

Data structures

Compiler passes

...

HipSpec

Coding

TIP

Hipster

Interactive Theorem Proving
Summary

- Progress in automated induction
  - rev-qrev
  - rotate-len
  - take-drop-len-rev-minus
- New benchmark suite
- Standardised benchmark format
- Set of tools to manipulate inductive problems