HipSpec

Automating Inductive Proofs of Program Properties

Dan Rosén

Koen Claessen, Moa Johansson, Nicholas Smallbone

Chalmers University of Technology | University of Gothenburg

July 1, 2012

What is HipSpec?

Haskell source

```
rev [] = []
rev (x:xs)
  = rev xs ++ [x]

prop_rev xs
  = rev (rev xs) =:= xs
```

What is HipSpec?

Hip

Haskell Inductive Prover

- FOL translation
- Apply induction
- Success

Haskell source

```
rev [] = []
rev (x:xs)
  = rev xs ++ [x]

prop_rev xs
  = rev (rev xs) =:= xs
```

What is HipSpec?

Haskell source

```
rev [] = []
rev (x:xs)
  = rev xs ++ [x]

prop_rev xs
  = rev (rev xs) =:= xs
```

Hip

Haskell Inductive Prover

- FOL translation
- Apply induction
- Success, or stuck!

What is HipSpec?

Haskell source

```
rev [] = []
rev (x:xs)
  = rev xs ++ [x]

prop_rev xs
  = rev (rev xs) =:= xs
```

Hip

Haskell Inductive Prover

- FOL translation
- Apply induction
- Success, or stuck!

QuickSpec

Eq-theory from testing:

```
rev (xs ++ ys)
  = rev ys ++ rev xs
xs ++ [] = []
xs ++ (ys ++ zs) =
  (xs ++ ys) ++ zs
```

What is HipSpec?

Haskell source

```
rev [] = []
rev (x:xs)
  = rev xs ++ [x]

prop_rev xs
  = rev (rev xs) =:= xs
```

Hip

Haskell Inductive Prover

- FOL translation
- Apply induction
- Success, or stuck!

QuickSpec

Eq-theory from testing:

```
rev (xs ++ ys)
  = rev ys ++ rev xs
xs ++ [] = []
xs ++ (ys ++ zs) =
  (xs ++ ys) ++ zs
```

HipSpec

Use these as lemmas!!

Example functional program and property

```
rev (x:xs) = rev xs ++ [x]
rev []     = []

qrev (x:xs) ys = qrev xs (x:ys)
qrev []     ys = ys
```

Goal: $\forall \texttt{xs}. \texttt{rev xs} = \texttt{qrev xs []}$

Structural induction

$$\frac{P(\texttt{[]}) \qquad \forall\, x, xs.\, P(xs) \implies P(x : xs)}{\forall\, xs.\, P(xs)}$$

Structural induction

$$\frac{P([]) \qquad \forall x, xs.\, P(xs) \implies P(x{:}xs)}{\forall xs.\, P(xs)}$$

lhs: `rev (x:xs)` = `rev xs ++ [x]`

rhs: `qrev (x:xs) []` = `qrev xs [x]`

Structural induction

$$\frac{P([\,]) \qquad \forall\, x, xs.\, P(xs) \implies P(x:xs)}{\forall\, xs.\, P(xs)}$$

lhs: `rev (x:xs)` = `rev xs ++ [x]`

rhs: `qrev (x:xs) []` = `qrev xs [x]`

hypothesis: `rev xs` = `qrev xs []`

Structural induction

$$\frac{P(\texttt{[]}) \qquad \forall\, x, xs.\, P(xs) \implies P(x\texttt{:}xs)}{\forall\, xs.\, P(xs)}$$

lhs: $\texttt{rev (x:xs)} = \texttt{rev xs ++ [x]} = \texttt{qrev xs [] ++ [x]}$

rhs: $\texttt{qrev (x:xs) []} = \texttt{qrev xs [x]}$

hypothesis: $\texttt{rev xs} = \texttt{qrev xs []}$

Structural induction

$$\frac{\mathsf{P}(\texttt{[]}) \qquad \forall\, \mathsf{x}, \mathsf{xs}.\, \mathsf{P}(\mathsf{xs}) \implies \mathsf{P}(\mathsf{x}:\mathsf{xs})}{\forall\, \mathsf{xs}.\, \mathsf{P}(\mathsf{xs})}$$

lhs: $\texttt{rev (x:xs)} = \texttt{rev xs ++ [x]} = \texttt{qrev xs [] ++ [x]}$

rhs: $\texttt{qrev (x:xs) []} = \texttt{qrev xs [x]}$

hypothesis: $\texttt{rev xs} = \texttt{qrev xs []}$

Stuck!!!

How do we proceed?

```
rev (x:xs) = rev xs ++ [x]
rev []     = []

qrev (x:xs) ys = qrev xs (x:ys)
qrev []     ys = ys
```

lhs: `rev (x:xs)` = `rev xs ++ [x]` = `qrev xs [] ++ [x]`

rhs: `qrev (x:xs) []` = `qrev xs [x]`

How do we proceed?

```
rev (x:xs) = rev xs ++ [x]
rev []     = []

qrev (x:xs) ys = qrev xs (x:ys)
qrev []     ys = ys
```

lhs: `rev (x:xs)` = `rev xs ++ [x]` = `qrev xs [] ++ [x]`

rhs: `qrev (x:xs) []` = `qrev xs [x]`

What about…

New Goal: $\forall$ xs, ys . `rev xs ++ ys` = `qrev xs ys`

How do we proceed?

```
rev (x:xs) = rev xs ++ [x]
rev []     = []

qrev (x:xs) ys = qrev xs (x:ys)
qrev []     ys = ys
```

lhs: `rev (x:xs)` $=$ `rev xs ++ [x]` $=$ `qrev xs [] ++ [x]`

rhs: `qrev (x:xs) []` $=$ `qrev xs [x]`

What about...

New Goal: $\forall$ `xs`, `ys` . `rev xs ++ ys` $=$ `qrev xs ys`

Then with `ys` $=$ `[]`, we get

`rev xs ++ []` $=$ `qrev xs []`

How do we proceed?

```
rev (x:xs) = rev xs ++ [x]
rev []     = []

qrev (x:xs) ys = qrev xs (x:ys)
qrev []     ys = ys
```

lhs: $\texttt{rev (x:xs)} = \texttt{rev xs ++ [x]} = \texttt{qrev xs [] ++ [x]}$

rhs: $\texttt{qrev (x:xs) []} = \texttt{qrev xs [x]}$

What about...

New Goal: $\forall\, \texttt{xs}, \texttt{ys}\,.\, \texttt{rev xs ++ ys} = \texttt{qrev xs ys}$

Then with $\texttt{ys} = \texttt{[]}$, we get

$$\texttt{rev xs} = \texttt{rev xs ++ []} = \texttt{qrev xs []}$$

The crucial lemma

New Goal: $\forall\, \texttt{xs}, \texttt{ys}.\ \texttt{rev xs ++ ys} = \texttt{qrev xs ys}$

Induction on xs

The crucial lemma

$$\text{New Goal:} \quad \forall\, \mathtt{xs}, \mathtt{ys}.\, \mathtt{rev\ xs\ ++\ ys} = \mathtt{qrev\ xs\ ys}$$

Induction on xs

Base, to show: $\forall\, \mathtt{ys}.\, \mathtt{rev\ []\ ++\ ys} = \mathtt{qrev\ xs\ []}$

$$\begin{aligned}\text{lhs:} &\quad \mathtt{rev\ []\ ++\ ys} = \mathtt{ys} \\ \text{rhs:} &\quad \mathtt{qrev\ []\ ys} = \mathtt{ys}\end{aligned}$$

The crucial lemma

$$\text{New Goal:} \quad \forall \mathtt{xs}, \mathtt{ys}.\, \mathtt{rev\ xs\ ++\ ys} = \mathtt{qrev\ xs\ ys}$$

Induction on $\mathtt{xs}$

Base, to show: $\forall \mathtt{ys}.\, \mathtt{rev\ []\ ++\ ys} = \mathtt{qrev\ xs\ []}$

$$\begin{aligned} \text{lhs:} &\quad \mathtt{rev\ []\ ++\ ys} = \mathtt{ys} \\ \text{rhs:} &\quad \mathtt{qrev\ []\ ys} = \mathtt{ys} \end{aligned}$$

Hooray!

The crucial lemma

Step, to show: $\forall\, ys.$ `rev (x:xs) ++ ys` $=$ `qrev (x:xs) ys`

Hypothesis: $\forall\, ys.$ `rev xs ++ ys` $=$ `qrev xs ys`

The crucial lemma

Step, to show: $\forall\, \texttt{ys}.\, \texttt{rev (x:xs) ++ ys} = \texttt{qrev (x:xs) ys}$

Hypothesis: $\forall\, \texttt{ys}.\, \texttt{rev xs ++ ys} = \texttt{qrev xs ys}$

$$
\begin{array}{lll}
\text{lhs} & = \texttt{rev (x:xs) ++ ys} & = \{\text{definition of rev}\} \\
 & \texttt{(rev xs ++ [x]) ++ ys} & = \{\text{associativity of ++}\} \\
 & \texttt{rev xs ++ ([x] ++ ys)} & = \{\text{definition of ++}\} \\
 & \texttt{rev xs ++ (x:ys)} & = \{\text{induction hypothesis on (x:ys)}\} \\
 & \texttt{qrev xs (x:ys)} & = \{\text{definition of qrev}\} \\
 & \texttt{qrev (x:xs) ys} & = \text{rhs}
\end{array}
$$

The crucial lemma

Step, to show: $\forall\, ys.\, \texttt{rev (x:xs) ++ ys} = \texttt{qrev (x:xs) ys}$

Hypothesis: $\forall\, ys.\, \texttt{rev xs ++ ys} = \texttt{qrev xs ys}$

$$
\begin{array}{lll}
\text{lhs} = & \texttt{rev (x:xs) ++ ys} & = \{\text{definition of } \texttt{rev}\} \\
& \texttt{(rev xs ++ [x]) ++ ys} & = \{\text{associativity of } \texttt{++}\} \\
& \texttt{rev xs ++ ([x] ++ ys)} & = \{\text{definition of } \texttt{++}\} \\
& \texttt{rev xs ++ (x:ys)} & = \{\text{induction hypothesis on } \texttt{(x:ys)}\} \\
& \texttt{qrev xs (x:ys)} & = \{\text{definition of } \texttt{qrev}\} \\
& \texttt{qrev (x:xs) ys} & = \text{rhs}
\end{array}
$$

HOORAY!

Success

We managed to prove

$$\forall \mathtt{xs}.\, \mathtt{rev\ xs} = \mathtt{qrev\ xs\ []}$$

Using:

- $\forall \mathtt{xs}, \mathtt{ys}.\, \mathtt{rev\ xs\ ++\ ys} = \mathtt{qrev\ xs\ ys}$
- Induction

Success

We managed to prove

$$\forall \mathtt{xs}.\, \mathtt{rev\ xs} = \mathtt{qrev\ xs\ []}$$

Using:

- $\forall \mathtt{xs}, \mathtt{ys}.\, \mathtt{rev\ xs\ ++\ ys} = \mathtt{qrev\ xs\ ys}$
- Induction
- But we also needed associativity of ++ and $\forall \mathtt{xs}.\, \mathtt{xs\ ++\ []} = \mathtt{xs}$, which need induction to be proved

Setting

Prove properties of functional programs using rewriting and induction.

Problem: Some of these properties require lemmas, that

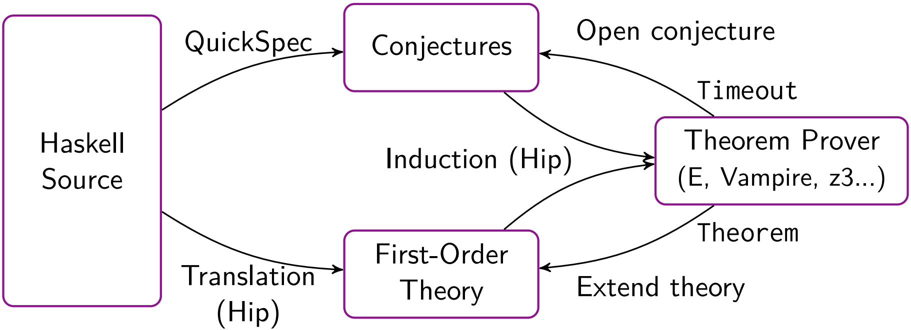
- Needs to be conjectured,
- Requires induction to be proved, and
- Might require lemmas themselves

Enter HipSpec

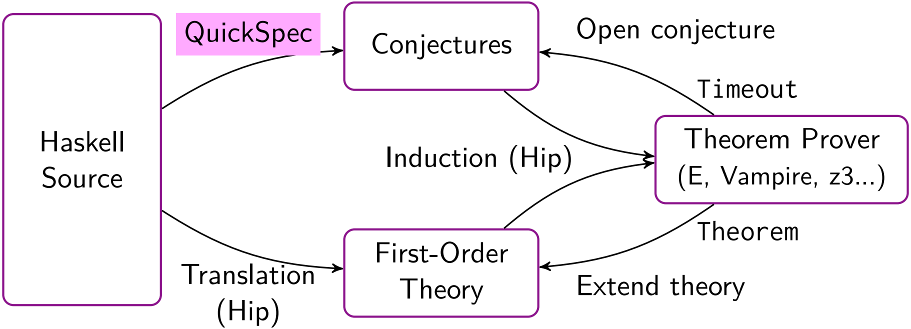
Solves this problems by:

- Generates an equational theory by counter-example testing,
- Try to prove this theory by applying induction
- Then, try to prove the user-stated properties
- Proof search with first-order theorem provers

Overview of HipSpec



Overview of HipSpec



Equivalence classes partitioning

Generates a bunch of terms:

`[]`	`[]++[]`	`qrev [] []`	`qrev (rev xs) []`
`qrev [] (rev xs)`	`qrev (rev xs) ys`	`qrev [] xs`	`qrev xs []`
`[]++qrev xs ys`	`qrev [] (xs++ys)`	`(x:xs)++[]`	`qrev xs ys++[]`
`qrev (x:[]) xs`	`qrev [] (x:xs)`	`rev []`	`rev (qrev ys xs)`
`rev (rev xs)`	`[]++rev xs`	`rev xs`	`rev xs++ys`
`xs`	`[]++xs`	`xs++[]`	`(xs++ys)++[]`
`[]++(xs++ys)`	`xs++ys`	`(x:[])++xs`	`xs++(x:[])`

Equivalence classes partitioning

```
xs
xs++[]
[]++xs
qrev [] xs
rev (rev xs)
qrev (rev xs) []
```

```
[]
rev []
qrev [] []
[]++[]
```

```
qrev xs ys
rev (qrev ys xs)
rev xs++ys
[]++qrev xs ys
qrev [] (qrev xs ys)
qrev xs ys++[]
qrev (qrev ys xs) []
```

```
xs++ys
qrev (rev xs) ys
[]++(xs++ys)
qrev [] (xs++ys)
(xs++ys)++[]
```

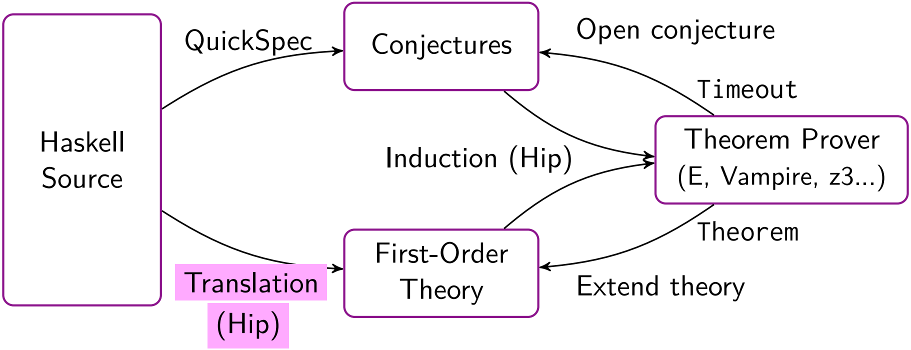
```
x:xs
[]++(x:xs)
qrev [] (x:xs)
(x:xs)++[]
(x:[])++xs
qrev (x:[]) xs
```

```
rev xs
qrev xs []
[]++rev xs
qrev [] (rev xs)
```

Example of pruned equations from QuickSpec

```
Universe has 2893 terms, 1824 classes
== equations ==
 1: xs++[] == xs
 2: qrev xs [] == rev xs
 3: []++xs == xs
 4: qrev [] xs == xs
 5: (x:xs)++ys == x:(xs++ys)
 6: (xs++ys)++zs == xs++(ys++zs)
 7: qrev xs ys++zs == qrev xs (ys++zs)
 8: qrev (x:xs) ys == qrev xs (x:ys)
 9: qrev (xs++ys) zs == qrev ys (qrev xs zs)
10: qrev (qrev xs ys) zs == qrev ys (xs++zs)
```

Overview of HipSpec



Hip : The Haskell Inductive Prover

- Translates the Haskell source definitions to first order logic

```
rev (x:xs) = rev xs ++ [x]
rev []     = []
```

Function definition axioms:

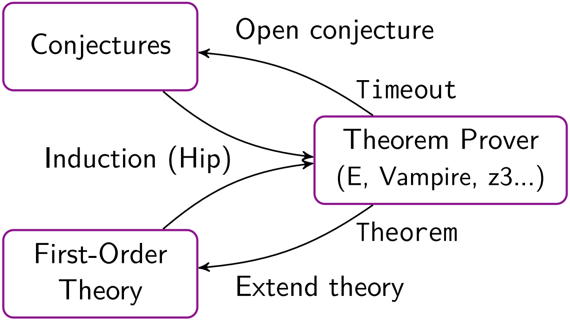
$$
\begin{array}{ll}
1 & \forall \mathsf{x}, \mathsf{xs}.\, \mathrm{rev}(\mathrm{cons}(\mathsf{x}, \mathsf{xs})) = \mathrm{append}(\mathrm{rev}(\mathsf{xs}), \mathrm{cons}(\mathsf{x}, \mathrm{nil})) \\
2 & \mathrm{rev}(\mathrm{nil}) = \mathrm{nil}
\end{array}
$$

Data type axioms:

$$
\begin{array}{ll}
3 & \forall \mathsf{x}, \mathsf{xs}, \mathsf{y}, \mathsf{ys}.\, \mathrm{cons}(\mathsf{x}, \mathsf{xs}) = \mathrm{cons}(\mathsf{y}, \mathsf{ys}) \implies \mathsf{x} = \mathsf{y} \wedge \mathsf{xs} = \mathsf{ys} \\
4 & \forall \mathsf{x}, \mathsf{xs}.\, \mathrm{nil} \neq \mathrm{cons}(\mathsf{x}, \mathsf{xs})
\end{array}
$$

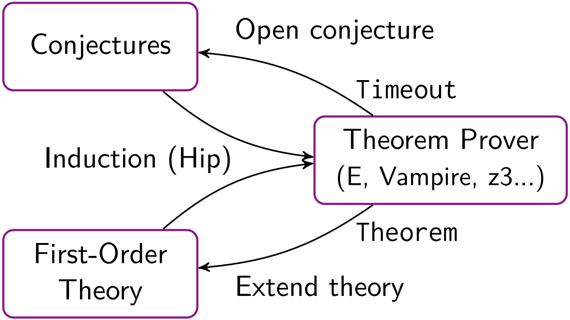
- Also supports higher-order functions and partial application
- Applies structural induction on properties

Picking a conjecture, and the main loop



1. Try to prove "smallest" unproved equation this round
2. Failure: save this for next round
3. Success: extend the theory
4. When a round did not lead to any successes, or everything proved, terminate.

Picking a conjecture, and the main loop



1. Try to prove "smallest" unproved equation this round
2. Failure: save this for next round
3. Success: extend the theory
4. When a round did not lead to any successes, or everything proved, terminate.

We use light-weight reasoning by means of a congruence closure to prune away conjecture that can be proved without induction.

Demo!

Evaluation - First Test Suite

First suite from *Case-Analysis for Rippling and Inductive Proof* by Johansson, Dixon and Bundy (2010)

85 conjectures, 71 equational.

Evaluation - First Test Suite

First suite from *Case-Analysis for Rippling and Inductive Proof* by Johansson, Dixon and Bundy (2010)

85 conjectures, 71 equational.

Tool	Proved conjectures (of 85)
Zeno	82
ACL2s	74
IsaPlanner	47
Dafny	45
HipSpec	67 (of 71)

Evaluation - First Test Suite

First suite from *Case-Analysis for Rippling and Inductive Proof* by Johansson, Dixon and Bundy (2010)

85 conjectures, 71 equational.

Tool	Proved conjectures (of 85)
Zeno	82
ACL2s	74
IsaPlanner	47
Dafny	45
HipSpec	67 (of 71)

Unproved:

```
count n xs = count n (sort xs),   len (filter p xs) <= len xs
sorted (sort xs) = True,          len (delete n xs) <= len xs
```

But they require conditional lemmas!

Evaluation - First Test Suite

First suite from *Case-Analysis for Rippling and Inductive Proof* by Johansson, Dixon and Bundy (2010)

85 conjectures, 71 equational.

Tool	Proved conjectures (of 85)
Zeno	82
ACL2s	74
IsaPlanner	47
Dafny	45
HipSpec	67 (of 71)

Two properties only proved by HipSpec!

```
rev (drop i xs) = take (len xs - i) (rev xs)
rev (take i xs) = drop (len xs - i) (rev xs)
```

Requires a bunch of quite far-fetched lemmas

Evaluation - Second Test Suite

Second test suite from *Productive Use of Failure in Inductive Proof* by Bundy and Ireland (1995)

Their tool CLAM supposedly proves all, but some properties contrived towards their tool, cf `rev (rev xs ++ []) = xs`

49 theorems, 38 equational.

Evaluation - Second Test Suite

Second test suite from *Productive Use of Failure in Inductive Proof* by Bundy and Ireland (1995)

Their tool CLAM supposedly proves all, but some properties contrived towards their tool, cf `rev (rev xs ++ []) = xs`

49 theorems, 38 equational. HipSpec proves 36!

Evaluation - Second Test Suite

Second test suite from *Productive Use of Failure in Inductive Proof* by Bundy and Ireland (1995)

Their tool CLAM supposedly proves all, but some properties contrived towards their tool, cf `rev (rev xs ++ []) = xs`

49 theorems, 38 equational. HipSpec proves 36!

Unproved:

No	Conjecture
T14	`ordered (isort xs) = True`
T50	`count x (isort xs) = count x xs`

Zeno?

Evaluation - Second Test Suite

Second test suite from *Productive Use of Failure in Inductive Proof* by Bundy and Ireland (1995)

Their tool CLAM supposedly proves all, but some properties contrived towards their tool, cf `rev (rev xs ++ []) = xs`

49 theorems, 38 equational. HipSpec proves 36!

Unproved:

No	Conjecture
T14	`ordered (isort xs) = True`
T50	`count x (isort xs) = count x xs`

Zeno? Proves 21/49

Success!

Success!?

There might be some limitations... ;)

Future work and current limitations

- Better heuristics (Equation order)
- Big theories and scalability
- Conditional properties
- Non-terminating programs and infinite values

Conclusion

Exploring the laws that hold through testing does not only help your understanding, but also helps to prove properties.

A form of completeness from QuickSpec: If there are laws up to a certain term size then QuickSpec is guaranteed to find them.

If the lemma is there, HipSpec will eventually try to prove it!

Extra slides

Obtaining HipSpec

- Clone the repository:
  `git clone http://github.com/danr/hipspec`
- Installation (requires GHC):
  ```
  cd hipspec
  git submodule update --init
  cabal install
  ```
- Install a theorem prover (say eprover)
- Try an example!
  ```
  cd testsuite/
  runghc Reverse.hs
  ```

Future work: Big theories

Taking all your functions from a big program:

- Testing takes a long time
- Lemmas become unrelated

Future work: Big theories

Taking all your functions from a big program:

- Testing takes a long time
- Lemmas become unrelated

How do we know when functions are related?

$$\texttt{length (xs ++ ys)} = \texttt{length (ys ++ xs)}$$

Future work: Conditional properties

Lemmas with implications:

$$\texttt{sorted xs} = \texttt{True} \implies \texttt{sorted (insert x xs)} = \texttt{True}$$

Future work: Conditional properties

Lemmas with implications:

$$\texttt{sorted xs} = \texttt{True} \implies \texttt{sorted (insert x xs)} = \texttt{True}$$

A trick: use a new data type, abstract for HipSpec:

```
data SortedList = SortedList { getSortedList :: [Nat] }

instance Arbitary SortedList where
  arbitrary = SortedList . scanl1 (+) `fmap` arbitrary
```

Future work: Conditional properties

Lemmas with implications:

$$\texttt{sorted xs} = \texttt{True} \implies \texttt{sorted (insert x xs)} = \texttt{True}$$

A trick: use a new data type, abstract for HipSpec:

```
data SortedList = SortedList { getSortedList :: [Nat] }
```

```
instance Arbitary SortedList where
  arbitrary = SortedList . scanl1 (+) `fmap` arbitrary
```

Now, we can state the property in terms of a sorted list `sl`:

$$\texttt{sorted (insert x (getSortedList sl))} = \texttt{True}$$

Need a notation to HipSpec that `SortedList` has a sorted invariant.

Proof: `rev (drop i xs) = take (len xs-i) (rev xs)`

No	Conjecture
1	`len (drop x xs) = len xs-x`
2	`len xs = len (rev xs)`
3	`xs = take x xs++drop x xs`
4	`rev (ys++xs) = rev xs++rev ys`
5	`xs = take (len xs) (xs++ys)`

```
rev (drop i xs)                                                   = {5}
take (len (rev (drop i xs))) (rev (drop i xs)++rev (take i xs))   = {2}
take (len (drop i xs)) (rev (drop i xs)++rev (take i xs))         = {1}
take (len xs-i) (rev (drop i xs)++rev (take i xs))                = {4}
take (len xs-i) (rev (take i xs++drop i xs))                      = {3}
take (len xs-i) (rev xs)
```

Future work: Conditional properties II

What about

$$\mathsf{x} \ < \ \mathsf{y} = \mathsf{True} \wedge \mathsf{y} \ < \ \mathsf{z} = \mathsf{True} \implies \mathsf{x} \ < \ \mathsf{z} = \mathsf{True}$$

Future work: Conditional properties II

What about

$$\mathtt{x} < \mathtt{y} = \mathtt{True} \land \mathtt{y} < \mathtt{z} = \mathtt{True} \implies \mathtt{x} < \mathtt{z} = \mathtt{True}$$

Same trick?

```
data Pair = Pair { smaller :: Nat , larger :: Nat }
```

Can we state the property?

$$\mathtt{smaller\ p1} < \mathtt{larger\ p2} = \mathtt{True}$$

Future work: Conditional properties II

What about

$$\mathtt{x} < \mathtt{y} = \mathtt{True} \wedge \mathtt{y} < \mathtt{z} = \mathtt{True} \implies \mathtt{x} < \mathtt{z} = \mathtt{True}$$

Same trick?

```
data Pair = Pair { smaller :: Nat , larger :: Nat }
```

Can we state the property?

$$\mathtt{smaller\ p1} < \mathtt{larger\ p2} = \mathtt{True}$$

Problem: how are p1 and p2 related?

Limitation: Expensive calculations

Imagine a program which does exponentiation, `**`, on unary nats
`data Nat = Zero | Succ Nat`

Too expensive to caluclate `x ** (y ** z)`.

Limitation: Expensive calculations

Imagine a program which does exponentiation, `**`, on unary nats
`data Nat = Zero | Succ Nat`

Too expensive to caluclate `x ** (y ** z)`.

```
f x y = if y == 1000000000000
          then 0
          else x + y
```