# Bounded Model Checking for Functional Programs

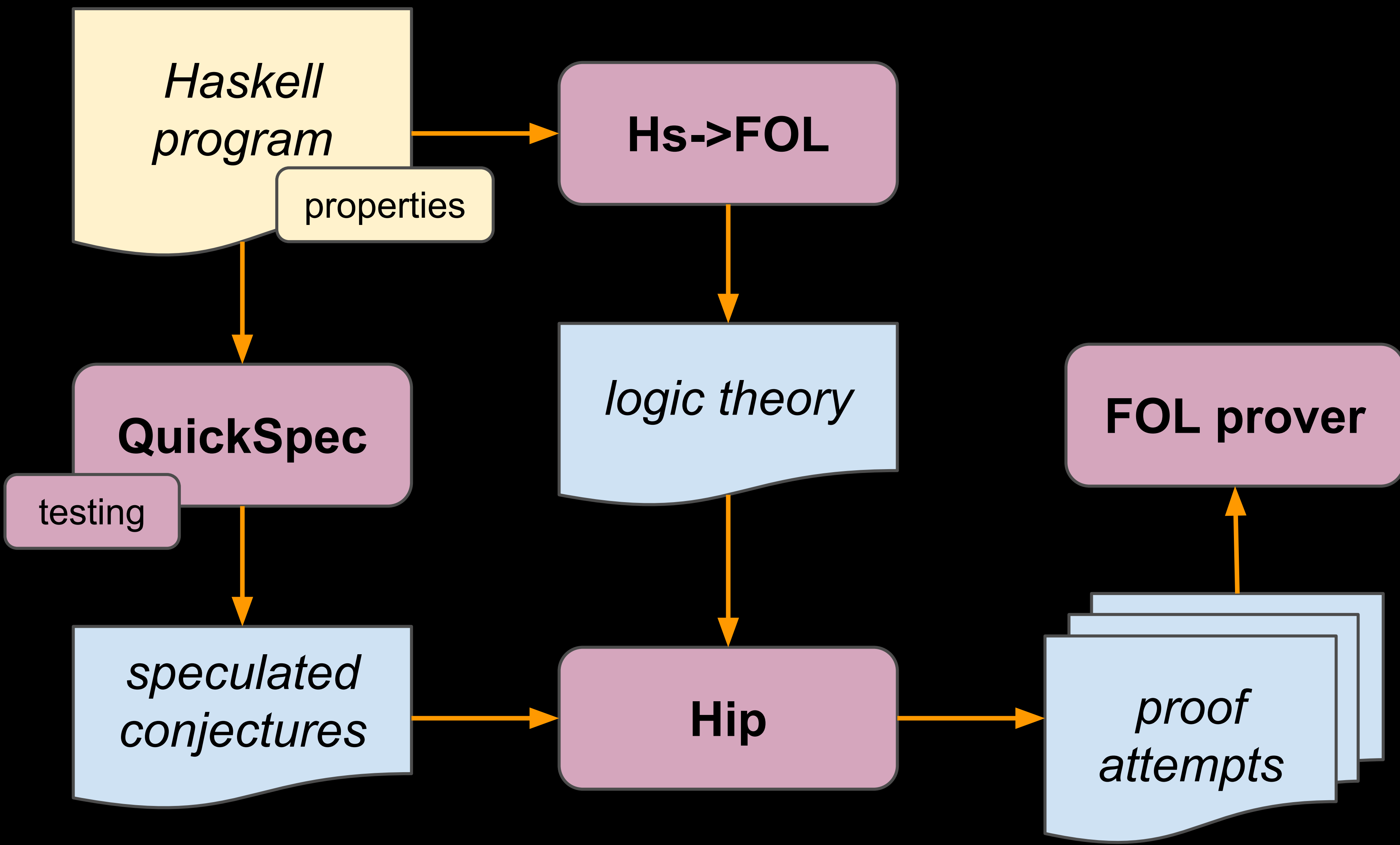
Koen Lindström Claessen
(joint work with Dan Rosén)

```
prop_Unambiguous t1 t2 =
  show t1 == show t2 ==> t1 == t2
```

```
prop_Unambiguous’ t1 t2 =
  t1 /= t2 ==> show t1 /= show t2
```

**very hard to test**

# HipSpec



Haskell program
properties
Hs->FOL
QuickSpec
testing
logic theory
FOL prover
speculated conjectures
Hip
proof attempts

# QuickSpec

```
          map f [] = []
    map f (map g xs) = map (f . g) xs
reverse (reverse xs) = xs
  map f (reverse xs) = reverse (map f xs)
                 ...
```

- automatically produced
- every equation is **tested** for correctness
- no equation is **logically implied*** by previous ones

?

```
sorted xs ==>
  sorted (insert x xs)
```

# TurboSpec

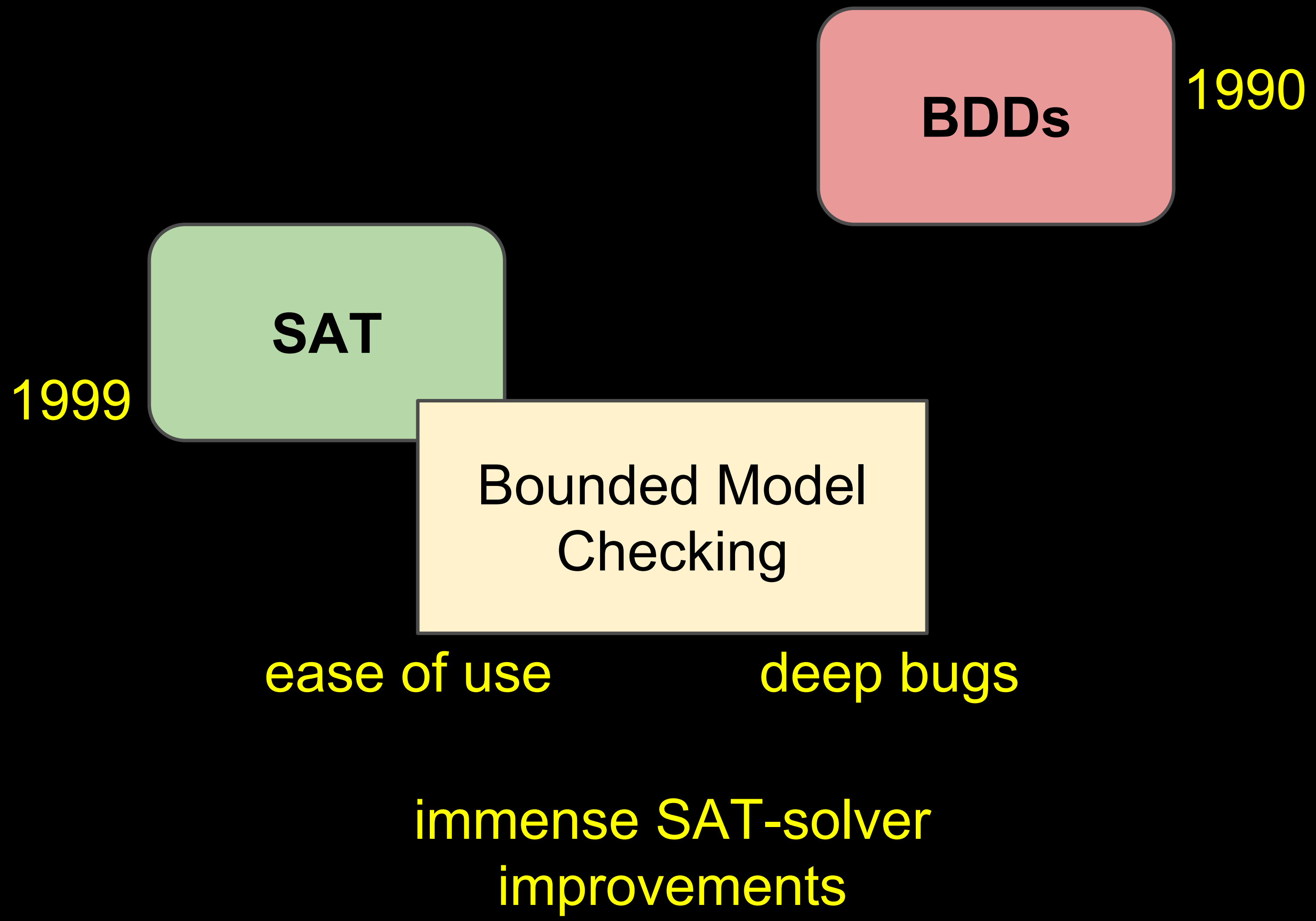
```
...
insert x xs = xs                ==> FALSE
insert x xs = insert y xs ==> x = y
insert x xs = insert x ys ==> xs = ys
...
```

```
sort xs = sort ys ==> sort xs = xs
                   \/ sort ys = ys
                   \/ xs = ys
```

**very hard to black-box test**

# The Problem

- Properties with
  - Strong pre-conditions
  - Weak post-conditions
  - No use of human intelligence
- How to find counter-examples?
- How to increase our confidence?


BDDs
1990
SAT
1999
Bounded Model Checking
ease of use
deep bugs
immense SAT-solver improvements

symbolic evaluation

bounded model checking

# Forte / FL

```
type Bool
```

```
var :: String -> Bool
```

```
FL> var “a” && not (var “a”)
False
```

```
FL> not (var “b” || not (var “a”))
a&&~b
```

```
FL> if var “a” then False else True
~a
```

```
FL> if var “a” then [] else [1]
ERROR: no booleans
```

```
type Prop
```

# Main Trick

```
data List a = Nil
            | Cons a (List a)
```

```
data Arg a = X | An a
```

```
data ListS a = NilCons Prop
                       (Arg a)
                       (Arg (ListS a))
```

decides whether Nil or Cons

# Symbolic If-Then-Else

```
if_then_else_ :: Prop -> Prop -> Prop -> Prop
```

```
if c then p else q = (c && p) || (not c && q)
```

# Symbolic If-Then-Else

```
if_then_else_ :: Prop -> Arg a -> Arg a -> Arg a
```

```
if c then X    else ab   = ab
if c then aa   else X    = aa
if c then An a else An b =
                           An (if c then a else b)
```

# Symbolic If-Then-Else

```
if_then_else_ :: Prop -> ListS a -> ListS a
                                 -> ListS a
```

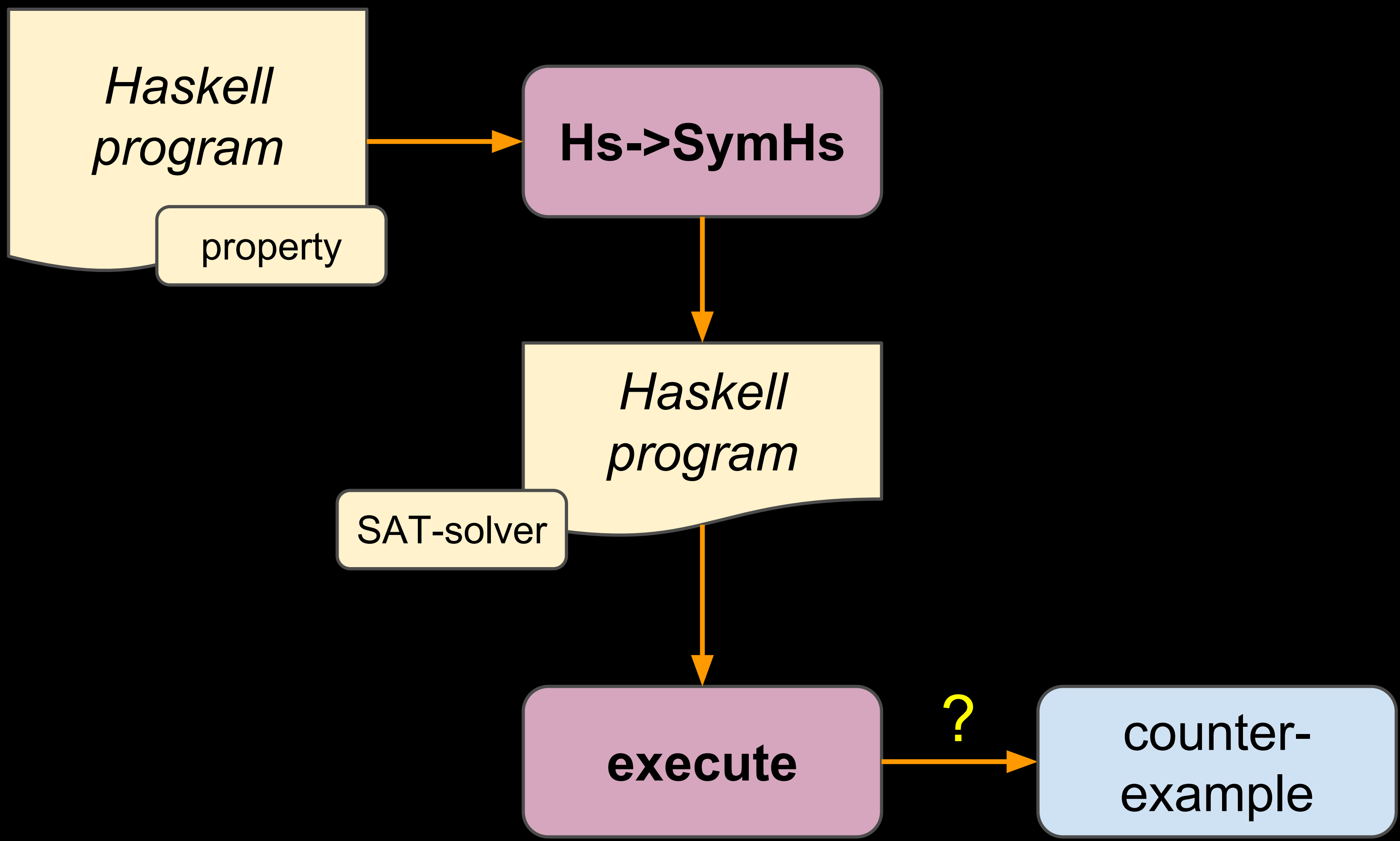
```
if c then NilCons p x xs else NilCons q y ys =
  NilCons (if c then p else q)
          (if c then x else y)
          (if c then xs else ys)
```

symbolic evaluation
on **bounded** inputs

unbounded inputs?

incrementality

# HBMC


Haskell program
property
Hs->SymHs
Haskell program
SAT-solver
execute
?
counter-example

```
vars :: Expr a -> [a]
vars (Var x)   = [x]
vars (Add a b) = vars a ++ vars b
vars (Neg a)   = vars a
```

```
vars e res =
  wait e $ \(Expr c ax aa ab) ->
    do when (c `is` Var) $
         singleton (un ax) res
       when (c `is` Add) $
         do va <- new
            vars (un aa) va
            vb <- new
            vars (un ab) vb
            append va vb res
       when (c `is` Neg) $
         do vars (un aa) res
```

# Generating Constraints

```
type C a -- Monad

newVar :: C Prop
insist :: Prop -> C ()
when   :: Prop -> C () -> C ()
```

```
when a (insist b) == insist (a => b)
when a (when b p) == when (a && b) p
when false p      == skip
```

# Finite Choice

```
type Fin a = [(Prop,a)]
```

```
newFin :: [a] -> C (Fin a)
newFin xs =
  sequence [ (x,) `fmap` newVar | x <- xs ]
```

```
is :: Eq a => Fin a -> a -> Prop
pxs `is` x = lookup x (pxs++[(x,false)])
```

```
vars :: Expr a -> [a]
vars (Var x)   = [x]
vars (Add a b) = vars a ++ vars b
vars (Neg a)   = vars a
```

```
vars e res =
  wait e $ \(Expr c ax aa ab) ->
    do when (c `is` Var) $
         singleton (un ax) res
       when (c `is` Add) $
         do va <- new
            vars (un aa) va
            vb <- new
            vars (un ab) vb
            append va vb res
       when (c `is` Neg) $
         do vars (un aa) res
```

# Incrementality

```
type Delay a

delay :: C a -> C (Delay a)
force :: Delay a -> C a
wait  :: Delay a -> (a -> C ()) -> C ()
```

# Example: Expr

```
data Expr a = Var a
            | Add (Expr a) (Expr a)
            | Neg (Expr a)
```

```
data ExprL   = Var | Add | Neg
data ExprC a = Expr (Fin ExprL)
                    (Arg a)
                    (Arg (ExprS a))
                    (Arg (ExprS a))
data ExprS a = Delay (ExprC a)
```

# Symbolic Expressions

```
class Constructive a where
  new :: C a
```

```
instance Constructive a =>
           Constructive (ListS a) where
  new = delay $
    do c  <- newFin [Nil,Cons]
       x  <- new  -- :: a
       xs <- new  -- :: ListS a
       return (ListS c (An x) (An xs))
```

```
vars :: Expr a -> [a]
vars (Var x)   = [x]
vars (Add a b) = vars a ++ vars b
vars (Neg a)   = vars a
```

```
vars e res =
  wait e $ \(Expr c ax aa ab) ->
    do when (c `is` Var) $
         singleton (un ax) res
       when (c `is` Add) $
         do va <- new
            vars (un aa) va
            vb <- new
            vars (un ab) vb
            append va vb res
       when (c `is` Neg) $
         do vars (un aa) res
```

# Translation of Programs

```
f x y = e
```

```
f x y res =
  <<e->res>>
```

```
<<f x y->res>> =
  f x y res
```

```
<<(let x = e1 in e2)->res>> =
  do x <- new
     <<e1->x>>
     <<e2->res>>
```

# Case Expressions

```
<<(case xs of
     Nil        -> e1
     Cons y ys -> e2 y ys)->res>>
```

```
wait xs (\(List c ay ays) ->
  do when (c `is` Nil)
       <<e1->res>>

     when (c `is` Cons)
       <<e2 (un ay) (un ays)->res>>
)
```

```
un :: Arg a -> a
un (An x) = x
```

```
vars :: Expr a -> [a]
vars (Var x)   = [x]
vars (Add a b) = vars a ++ vars b
vars (Neg a)   = vars a
```

```
vars e res =
  wait e $ \(Expr c ax aa ab) ->
    do when (c `is` Var) $
         singleton (un ax) res
       when (c `is` Add) $
         do va <- new
            vars (un aa) va
            vb <- new
            vars (un ab) vb
            append va vb res
       when (c `is` Neg) $
         do vars (un aa) res
```

# Example

```
prop_NoPalindromes (xs::[Bool]) =
  length xs >= 3 ==>
    reverse xs /= xs
```

# Call merging

```
case t of
  Empty ->
  Node a p q | x < a       -> ... f p ...
             | otherwise -> ... f q ...
```

```
let pq =
       case t of
         Node a p q | x < a       -> p
                    | otherwise -> q
 in case t of
       Empty ->
       Node a p q | x < a       -> ... f pq ...
                  | otherwise -> ... f pq ...
```

# Main Solving Loop

1. Generate initial constraints by executing the program

2. Solve, assuming that no waiting computation can happen

3. If solution, then done

4. Otherwise, pick one waiting computation, force it, and go to 3.

```
vars :: Expr a -> [a]
vars (Var x)   = [x]
vars (Add a b) = vars a ++ vars b
vars (Neg a)   = vars a
```

```
vars e res =
  wait e $ \(Expr c ax aa ab) ->
    do when (c `is` Var) $
         singleton (un ax) res
       when (c `is` Add) $
         do va <- new
            vars (un aa) va
            vb <- new
            vars (un ab) vb
            append va vb res
       when (c `is` Neg) $
         do vars (un aa) res
```

# Memoization

- In symbolic evaluation…
- … **all branches** of a case are executed!
- Functions are applied much more often…
- … and more often to **the same arguments** multiple times!

# Which Wait to Force?

- If no solution, then we have a **conflict** …
- … a subset of the assumptions that is contradictory
- Keep a **queue** of waiting computations …
- … and always expand the computation that is part of the conflict that is most to the head of the queue

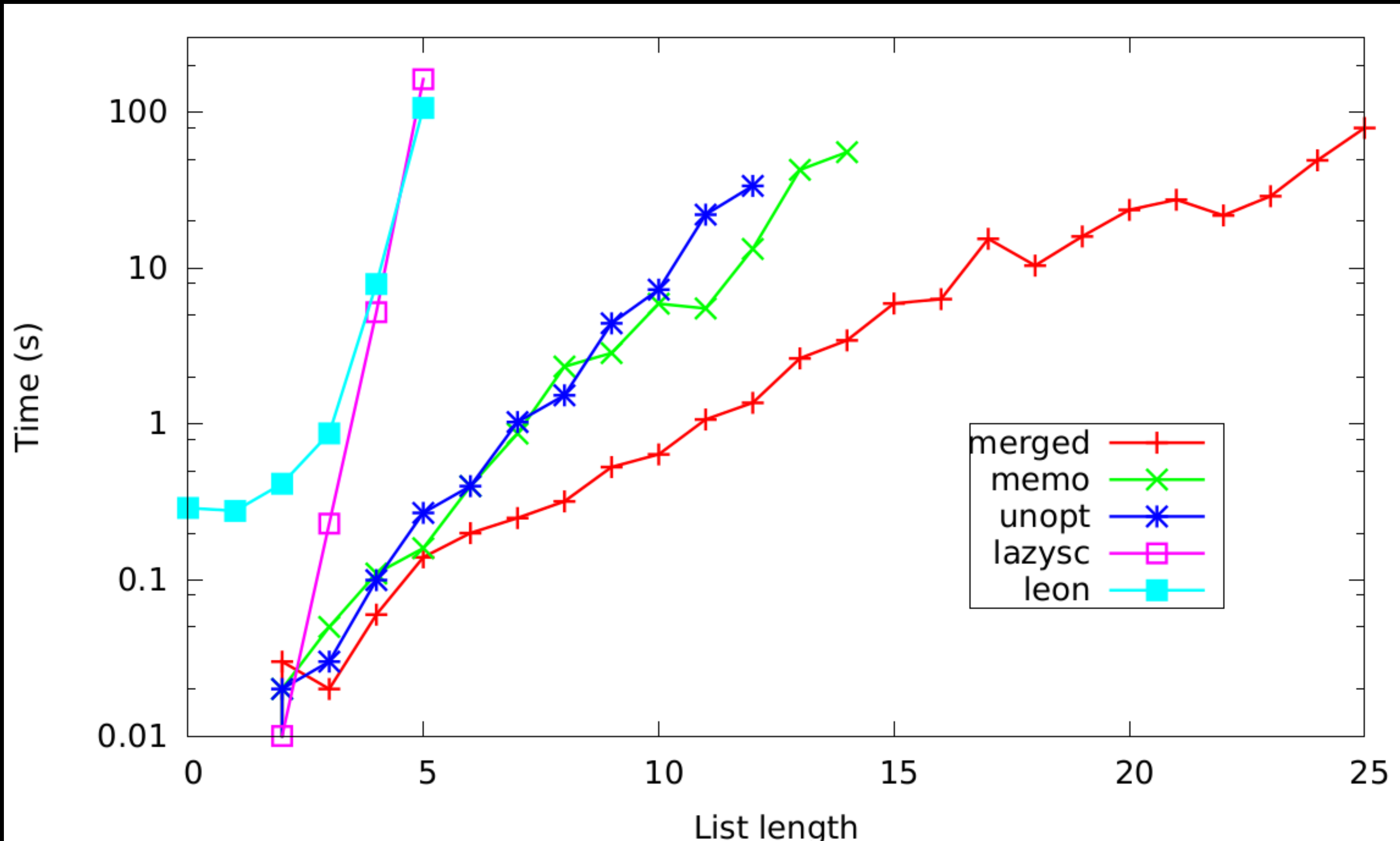
# Example: Usorted list

```
usorted :: [Nat] -> Bool
usorted (x:y:xs) = x < y && usorted (y:xs)
usorted _        = True
```

```
xs: _
xs: Lst__
xs: Lst_(Lst__)
xs: Lst_(Lst_(Lst__))
xs: Lst_(Lst_(Lst_(Lst__)))
xs: Lst_(Lst(Nat_)(Lst_(Lst__)))
xs: Lst_(Lst(Nat_)(Lst(Nat_)(Lst__)))
xs: Lst(Nat_)(Lst(Nat_)(Lst(Nat_)(Lst__)))
xs: Lst(Nat_)(Lst(Nat(Nat_))(Lst(Nat_)(Lst__)))
xs: Lst(Nat_)(Lst(Nat(Nat_))(Lst(Nat(Nat_))(Lst__)))
xs= [Z,S Z,S (S Delayed_Nat)]
```

# Example: Merge

```
merge :: Ord a => [a] -> [a] -> [a]
merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) =
  let (a,as,bs)
       | x <= y    = (x, xs,   y:ys)
       | otherwise = (y, x:xs, ys)
   in a : merge as bs
```



Time (s)
100
10
1
0.1
0.01
0
5
10
15
20
25
List length
merged
memo
unopt
lazysc
leon

# Example: Turing Machine

```
data Sym    = A | B | O
type State  = Nat
data Action = Lft State | Rgt State | Stp
```

```
type Q = (State,Sym) -> (Sym,Action)
```

```
run :: Q -> [Sym] -> [Sym]
```

# Termination

- Some functions may not terminate
- (even though their non-symbolic versions do terminate!)
- In such cases, we introduce an artificial wait

```
postpone :: C () -> C ()
postpone p =
  do x <- new -- :: ()
     wait x $ \() -> p
```

# Example: Turing Machine

```
cond q =
  run q [A]               == [A] &&
  run q [B,A,A,A,A,B] == [A,A,A,A,B,B]
```

```
(0,A) -> (A,Stop)
(0,B) -> (A,Rgt 1)
(1,A) -> (A,Rgt 1)
(1,B) -> (B,Lft 2)
(2,A) -> (B,Stop)
```

# Other examples

- Type checker
  -> find terms of a certain type

- Regular expression recognizer
  -> find bugs in recognizer
  -> find buggy laws

- Grammar specification
  -> natural language ambiguities

# Current Work

- SMT
  - Integer theory
  - Equality / functions (Leon)
- Improve incrementality
  - Conflict minimization
- Memory use / garbage collection
- TurboSpec

# Conclusions

- HipSpec = QuickSpec + Hip
- Speculating conjectures needs smart ways of finding counter examples
- Using SAT is one such way
- Benchmark suite for
  - Automated induction problems
  - False properties

https://github.com/tip-org