In Search of a Suitable Induction Principle for Automated Induction

Koen Claessen

2018

joint work with

Linnea Andersson and Andreas Wahlstöm

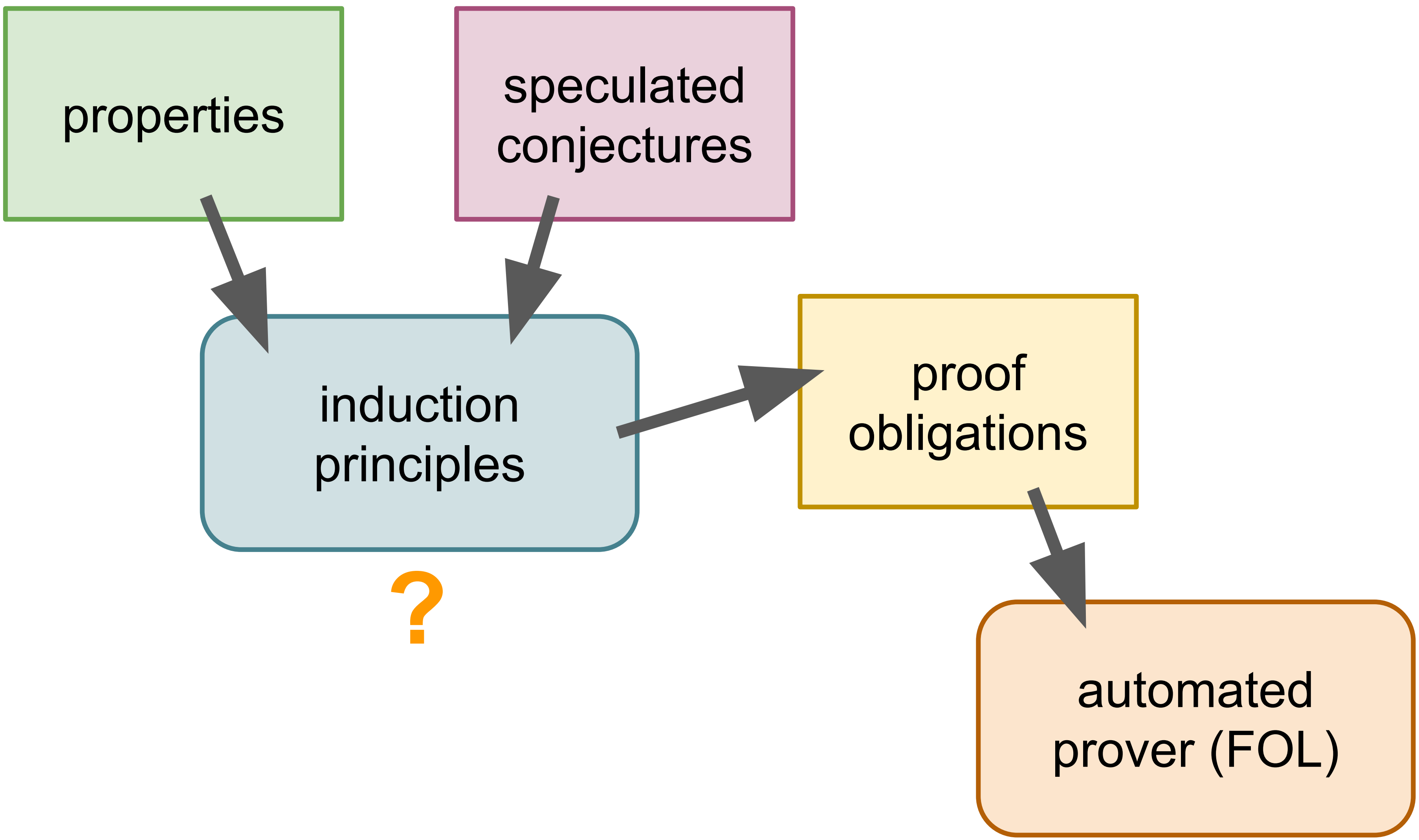
```
quicksort []     = []
quicksort (x:xs) =
  quicksort [ y | y <- xs, y <= x ]
  ++ [x] ++
  quicksort [ y | y <- xs, y > x ]
```

```
ordered []       = True
ordered [x]      = True
ordered (x:y:xs) = x <= y && ordered (y:xs)
```

```
∀xs . ordered (quicksort xs)
```

?

HipSpec / Hipster / TurboSpec / ...



induction principle

structural

`∀xs . ordered (quicksort xs)`

φ

`∀xs,ys,n . P(xs,ys,n)`

- structural induction over xs
- str… over ys
- … over n
- …ral induction over xs,n
- …ral induction over xs,ys,n

BAD:
- not enough

GOOD:
- simple
- often works
- reasonable amount of possibilities

```haskell
quicksort []     = []
quicksort (x:xs) =
  quicksort [ y | y <- xs, y <= x ]
  ++ [x] ++
  quicksort [ y | y <- xs, y > x ]
```

```haskell
ordered []       = True
ordered [x]      = True
ordered (x:y:xs) = x <= y && ordered (y:xs)
```

```
∀xs . ordered (quicksort xs)
```

?

induction principle

size-based

```
∀xs . ordered (quicksort xs)
```

```
∀xs,n . size xs = n ==>
         ordered (quicksort xs)
```

- structural induction over n

what is size?

let the prover search for size...

GOOD:
- works in many cases

```
∀x,n . size x =
```

BAD:
- too many possibilities
- re-doing termination proof

- structural induction over n

induction principle

- powerful enough
- limited enough

recursion

induction principle

∀xs . ordered (quicksort xs)

```
quicksort []     = []
quicksort (x:xs) =
  quicksort [ y | y <
  ++ [x] ++
  quicksort [ y
```

GOOD:

- works in many cases

BAD:

- too many possibilities
- automation?

restrict/ specialize

```
         ( Q([ y | y <- xs, y <= x ]) &
Q([])      Q([ y | y <- xs, y > x ]) ) ==> Q(x:xs)
-------------------------------------------------
                ∀xs . Q(xs)
```

fixpoint

induction principle

```
∀xs . ordered (quicksort xs)
```

```
quicksort []     = []
quicksort (x:xs) =
  quicksort [ y | y <- xs, y <= x ]
  ++ [x] ++
  quicksort [ y | y <- xs, y > x ]
```

```
quicksort = H(quicksort)
```

$$\frac{P(\bot) \qquad P(f) \Longrightarrow P(H(f))}{P(\text{quicksort})}$$

GOOD:
- works with the program/function directly

BAD:
- non-termination
- brittle

application

induction principle

```
∀xs . ordered (quicksort xs)
```

```
P2(ys,b) =
  ∀xs . ( quicksort xs = ys &
          ordered ys = b ) ==> b
```

```
P1(xs,ys) =
  quicksort xs = ys ==> ordered ys
```

```
Q1(xs) = P1(xs,quicksort(xs))
```

use recursion induction

```
Q2(ys) = P2(ys,ordered(ys))
```

application

induction principle

```
∀xs . ordered (quicksort xs)
```

```
(as = [] & quicksort(as) = [])
\/
(as = b:bs & quicksort(as) =
               quicksort([y|y<-bs,y<=b])
               ++ [b] ++
               quicksort([y|y<-bs,y>b])
 & ordered(quicksort([y|y<-bs,y<=b]))
 & ordered(quicksort([y|y<-bs,y>b])))
 )
```

```
ordered(quicksort(as))
```

?

application

induction principle

```
∀xs . ordered (quicksort xs)
```

```
(bs = [] & ordered(bs) = True)
\/
(bs = [c] & ordered(bs) = True)
\/
(bs = c:d:cs & ordered(bs) =
                 c<=d && ordered(d:cs)
```

```
 & (∀xs . quicksort(xs)=d:cs ==>
              ordered(d:cs)))
```

```
∀xs . quicksort(xs)=bs ==>
ordered(bs)
```

?

application induction

GOOD:
- works in many cases
- works with the program/function directly
- helps the automated prover with instances

BAD:
- not enough?

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)

flatten1 :: Tree a -> [a]
flatten1 (Leaf x)   = [x]
flatten1 (Node v w) = flatten1 v ++ flatten1 w

flatten2 :: Tree a -> [a] -> [a]
flatten2 (Leaf x)   xs = x:xs
flatten2 (Node v w) xs = flatten2 v (flatten2 w xs)

flatten3 :: [Tree a] -> [a]
flatten3 []                = []
flatten3 (Leaf x   : ts) = x:flatten3 ts
flatten3 (Node v w : ts) = flatten3 (v:w:ts)
```

∀t . flatten3 [t] = flatten1 t ?

can we **replace** structural induction with application induction in real benchmarks?

TIP results

similar number of cases

everything that can be proved using **structural induction** can be proved using **application induction**

sometimes proofs are not found in time, **in practice**

surprising?

treat = as a recursive function

not for mutual recursion

sometimes non-trivial **application induction** instances are needed

```
even, odd :: Nat -> Bool
even Zero      = True
even (Succ n) = not (odd n)

odd Zero       = False
odd (Succ n) = not (even n)
```

```
∀b . even (Succ (Succ n)) = even n
```

?

unfolding

“deep” application induction

when proving a property about even...

```
even, odd :: Nat -> Bool
even Zero     = True
even (Succ n) = not (odd n)

odd Zero     = False
odd (Succ n) = not (even n)
```

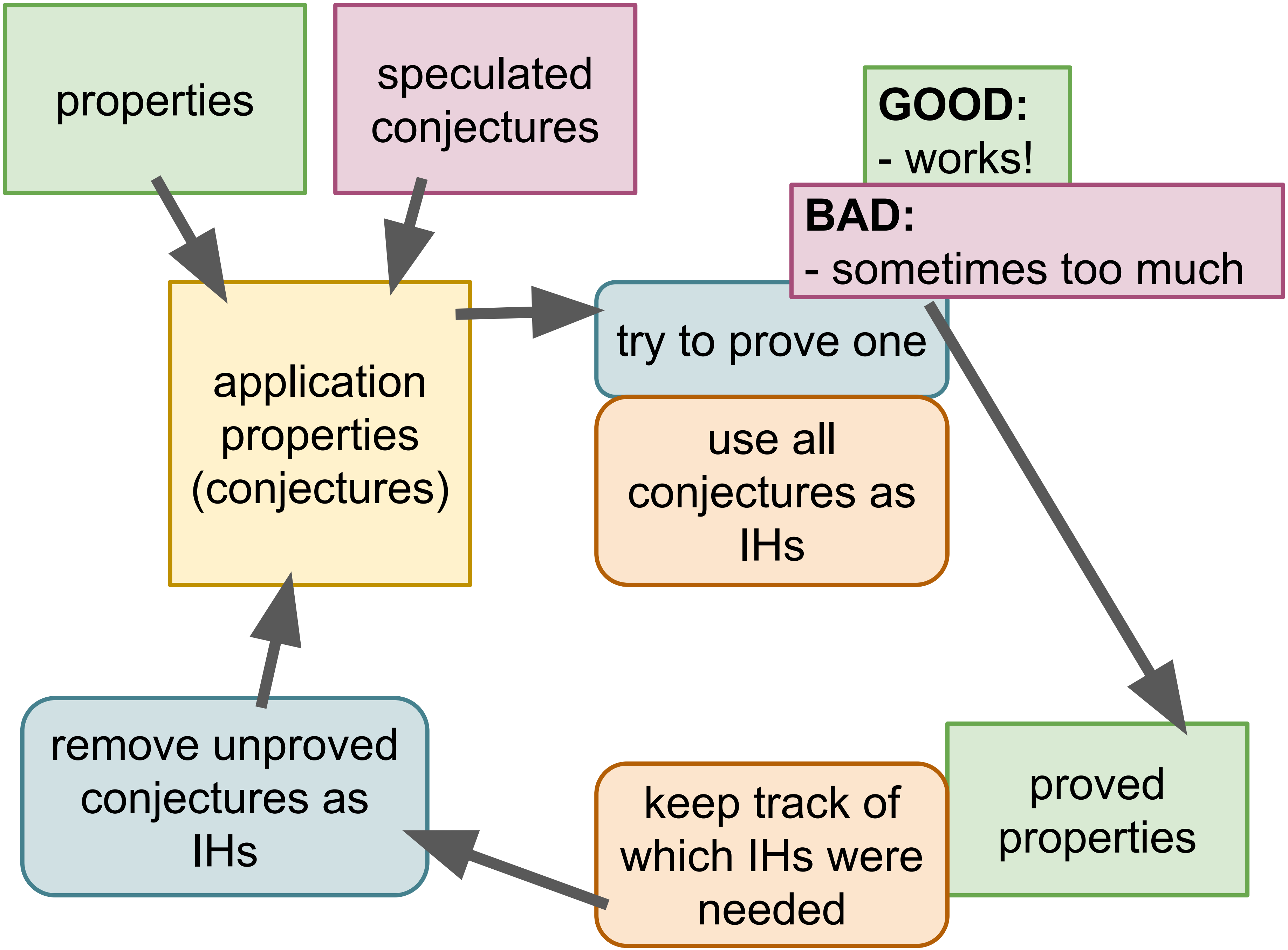
...assume it here

```
f = ... f ... g ...

g = ... g ...
```

prove g first

```
f = ... f ... g ...

g = ... f ... g ...
```

~~un ng~~


properties
speculated conjectures
GOOD:
- works!
BAD:
- sometimes too much
application properties (conjectures)
try to prove one
use all conjectures as IHs
proved properties
keep track of which IHs were needed
remove unproved conjectures as IHs

Summary + Conclusions

- Application induction can replace structural induction in practice
  - similar number of cases to try
  - also subsumes recursion induction in practice
- Mutual recursion needs to improve
  - dependency analysis?
- Integrate properly with TurboSpec
  - using counter-examples for conjectures