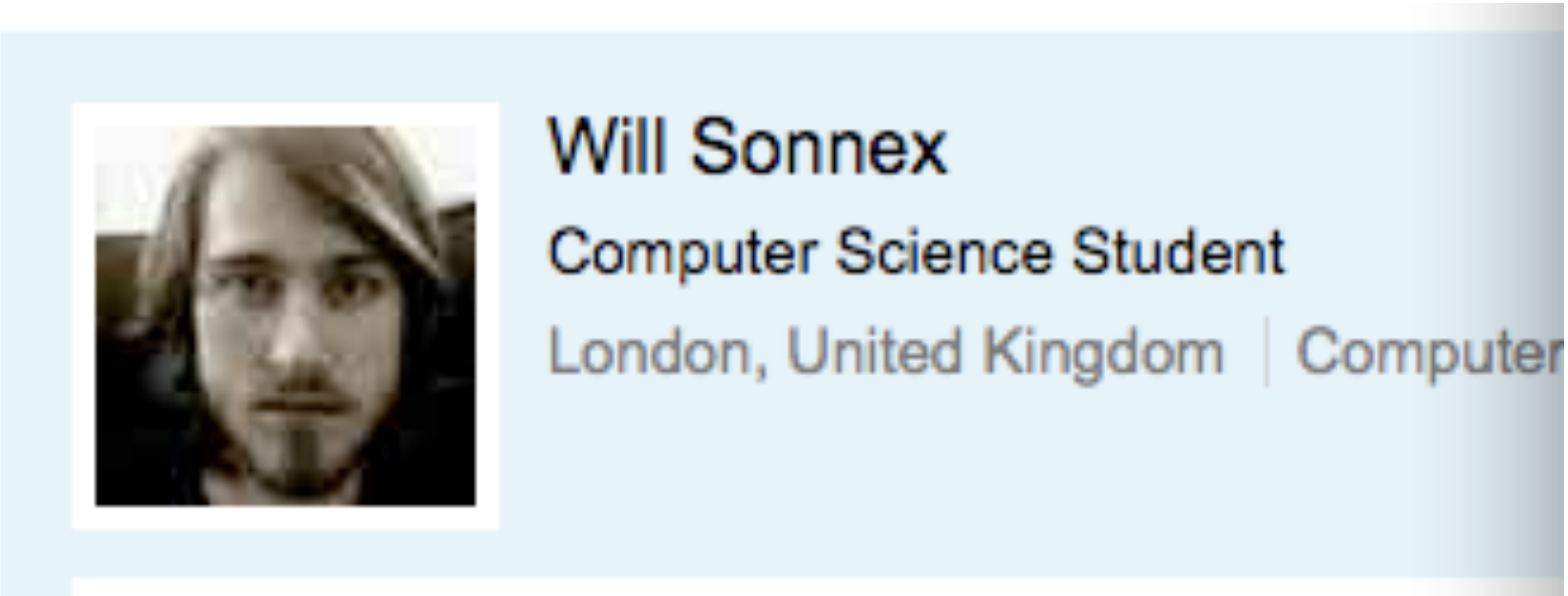
- [Zeno of Elea] (c.490–c.430 BC), philosopher, follower of Parmenides, famed for his *paradoxes*.
- [Zeno of Citium] (333 BC - 264 BC), founder of the Stoic school of philosophy
- [Zeno of Tarsus] (200s BC), Stoic philosopher
- [Zeno of Sidon] (1st century BC), Epicurean philosopher
- Zeno at http://www.haskell.org/haskellwiki/Zeno

Zeno

an automated theorem prover for properties of inductive structures



Will Sonnex,

Sophia Drossopoulou and Susan Eisenbach

Zeno

- Proves equality over Haskell-like expressions of the form
  $E_1 = E_2, \ldots, E_{2n+1} = E_{2n+2} \implies E = E'$
  where `E` may mention recursively defined functions
- Zeno can prove properties like
  - `rev (rev xs) = xs`
  - `order (order xs) = order xs`
  - `mult x (succ 0) = x`
- Variables implicitly universally quantified; no existentials
- Booleans are encoded through the `Bool` data type.
- Zeno can discover necessary auxiliary lemmas.
- Zeno cannot use theories.

Zeno

- Proves equality over Haskell-like expressions of the form
  `E1=E2,…,E2n+1=E2n+2   ==> E = E'`
  where `E` may mention recursively defined functions
- Zeno can prove properties like
  - `rev (rev xs) = xs`
  - `order (order xs) = order xs`
  - `mult x (succ 0) = x`
- Variables implicitly universally quantified; no existentials.
- Booleans encoded through the `Bool` data type.
- Zeno can discover necessary auxiliary lemmas.
- Zeno cannot use theories.
- **From a benchmark suite suggested by Isaplanner, Zeno can prove more properties than Isaplanner and ACL2s**

… using the Isaplanner test suite

Theorem prover	Percentage proven	Identifiers of unproven properties
Dafny (Z3 and IND)	53.5%	45-85
Isaplanner	55%	47-85
ACL2s – coded types	87%	47, 50, 54, 56, 72, 73, 74, 81, 83, 84, 85
Zeno	96%	72, 74, 85

This Talk

- Example Zeno code
- The proof steps – by example
- Trimming the search space

Example - Haskell

```
data Nat = Zero | Succ Nat

(<=) :: Nat -> Nat -> Bool
Zero <= _  = True
Succ x <= Zero = False
Succ x <= Succ y = x <= y
```

Example - Haskell

```haskell
data Nat = Zero | Succ Nat

(<=) :: Nat -> Nat -> Bool
Zero <= _ = True
Succ x <= Zero = False
Succ x <= Succ y = x <= y

srtd :: [Nat] -> Bool
srtd [] = True
srtd [x] = True
srtd (x:y:zs) = (x <= y) && srtd (y:zs)
```

Example - Haskell

```
data Nat = Zero | Succ Nat

(<=) :: Nat -> Nat -> Bool
Zero <= _ = True
Succ x <= Zero = False
Succ x <= Succ y = x <= y

srtd :: [Nat] -> Bool
srtd [] = True
srtd [x] = True
srtd (x:y:zs) = (x <= y) && srtd (y:zs)

ordr :: [Nat] -> [Nat]
ordr [] = []
ordr (x:xs) = ins x (ordr xs)
```

Example - Haskell

```
data Nat = Zero | Succ Nat

(<=) :: Nat -> Nat -> Bool
Zero <= _ = True
Succ x <= Zero = False
Succ x <= Succ y = x <= y

srtd :: [Nat] -> Bool
srtd [] = True
srtd [x] = True
srtd (x:y:zs) = (x <= y) && srtd (y:zs)

ordr :: [Nat] -> [Nat]
ordr [] = []
ordr (x:xs) = ins x (ordr xs)

ins :: Nat -> [Nat] -> [Nat]
ins n [] = [n]
ins n (x:xs) | n<=x = n:x:xs | otherwise x:(ins n xs)
```

This Talk

- Example Zeno code
- The proof steps – by example
- Trimming the search space

Zeno supports sequent-style proof rules.
It applies these rules backwards, possibly trying several.
These rules are:

- CON - contradiction
- EQL – substitute equals for equals
- IND - induction
- EXP – expansion
- GEN – generalization
- CASE – case analysis
- Modus Ponens

So, we want to prove

```
srtd (ordr is)
```

We will first outline part of the proof, and then we will show the rules for the individual steps.

So, we want to prove

```
srtd (ordr is)
```

We will first outline part of the proof, and then we will show the rules for the individual steps.

Proving `srtd (ordr is)`

```
                    ????
------------------------------------------ ???
              srtd (ordr is)
```

Proving `srtd (ordr is)` - induction

```
      ???                            ????
______________ ???    _______________________________________ ????
srtd (ord [])         srtd (ord js) => srtd (ord j:js)
_________________________________________________________ IND
                     srtd (ordr is)
```

Proving `srtd (ordr is)` - definition

```
    ???                        ????
_____________???   ______________________________________????
srtd (ord [])      srtd (ord js) => srtd (ord j:js)
______________________________________________________IND
                 srtd (ordr is)
```

Proving srtd (ordr is)

```
                                  ????
                    ___________________________________________ ???
                      srtd (ord js) => srtd (ins j (ord js))
      ???
_____________ ???   ___________________________________________ EXP
srtd (ord [])         srtd (ord js) => srtd (ord j:js)
_______________________________________________________________ IND
                      srtd (ordr is)
```

Proving `srtd (ordr is)`

```
                                       ????
                  ------------------------------------------------ ???
                  srtd (ord js) => srtd (ins j (ord js))
     ???
------------------ ???    ------------------------------------------------ EXP
srtd (ord [])             srtd (ord js) => srtd (ord j:js)
 -------------------------------------------------------------------------- IND
                          srtd (ordr is)
```

Proving `srtd (ordr is)` - generalization

```
________________________________________________________ ???
              srtd (ks) => srtd (ins i ks)
            ______________________________________ GEN
             srtd (ord js) => srtd (ins j (ord js))
.    ???
______________ ???   _____________________________ EXP
srtd (ord [])        srtd (ord js) => srtd (ord j:js)
 .
________________________________________________________ IND
                  srtd (ordr is)
```

Proving `srtd (ordr is)`

Note:
Zeno discovered the auxiliary lemma

`srtd ks  => srtd (ins j ks)`

```
 ------------------------------------------------------------------???
                      srtd (ks) => srtd (ins j ks)
                    ---------------------------------------------GEN
                   srtd (ord js) => srtd (ins j (ord js))
.     ???
--------------???  ----------------------------------------------EXP
srtd (ord [])       srtd (ord js) => srtd (ord j:js)
 .
 ----------------------------------------------------------------IND
                       srtd (ordr is)
```

Proving `srtd (ordr is)` - induction

```
                                                         ???
                                         ___________________________________________________???
         ????                                srtd (ms)=> srtd (ins i (ord ms))
__________________________???                              =>
srtd ([])                                     srtd (m:ms)=> srtd (ins i (ord m:ms))
   => srtd (ins i [])
___________________________________________________________________________________IND
                                     srtd (ks) => srtd (ins i ks)
                               ____________________________________________________GEN
                                  srtd (ord js) => srtd (ins j (ord js))
.     ???                          ________________________________________________EXP
____________________???           srtd (ord js) => srtd (ord j:js)
srtd (ord [])
 .
__________________________________________________________________________________IND
                                       srtd (ordr is)
```

This Talk

- Example Zeno code
- The proof steps – by example
- Trimming the search space

Zeno’s trimming heuristics

- Prioritize CON and EQL steps.
- Search for counterexample.
- Critical expressions.
- Critical paths.
- ….

Zeno's trimming heuristics

- Prioritize CON and EQL steps.
  - CON and EQL "close" proof braches;

$$\frac{\texttt{K, K' are constructors} \quad \texttt{K =/= K'}}{\vdash (\texttt{K } E_1 \ldots E_n) = (\texttt{K' } E'_1 \ldots E'_n) \Rightarrow \phi}\ \texttt{CON}$$

therefore it pays to apply them ASAP

- Search for counterexample.
- Critical expressions.
- Critical paths.
- ….

Zeno’s trimming heuristics

- Prioritize CON and EQL steps.
- Search for counterexample.
  - After generation of new proof goal (eg through GEN), create examples (using critical expressions/paths) and discard the branch if counterexample found.
- Critical expressions.
- Critical paths.
- ….

Zeno's trimming heuristics

- Prioritize CON and EQL steps.
- Search for counterexample after GEN steps.
- Critical expressions.
  - Aim to steer the proof search so that EXP steps become applicable (ie function definitions may be applied).
- Critical paths.
- ….

Zeno's trimming heuristics

- Prioritize CON and EQL steps.
- Search for counterexample after GEN steps.
- Critical expressions.
  - Aim to steer the proof search so that EXP steps become applicable (ie function definitions may be applied).
    This is in contrast with rippling (Isaplanner), which, instead, tries to make the inductive hypothesis applicable.
- Critical paths.
- ….

Critical expressions - example

???

srtd (ordr is)

???

Critical expressions - example

At this point, many steps are applicable:

- `IND` on `is`
- `CASE` on `ord is`
- `CASE` on `srtd(ord is)`
- `IND` on `ord is`
- `CASE` on `first(is)`
- …

```
                                                  ???
srtd (ordr is)
```

Critical expressions - example

Similarly, at this point, the following steps are applicable:

- `IND` on `js`
- `IND` on `j`
- `CASE` on `js`
- `CASE` on `j`
- `CASE` on `ord js`
- `CASE` on `ord j:js`
- …

```
                          ????
                     ----------------------------------------------????
 ...                  srtd (ord js) => srtd (ord j:js)
 -------------------------------------------------------------------IND
                          srtd (ordr is)
```

Critical expressions - definition

We want to consider only those expressions which are critical for the execution of the term, ie those expressions where execution of a term will get stuck.

Crits(E) =

- E — if E is normal
- E′ — if E->* **case** E′ **of** …, E′ ∉ E
- Crits(E′) — if E->* **case** E′ **of** …, E′ ∈ E

E is *normal* if it cannot be further re-written

Critical expressions - examples

$$\text{Crits}(E) = \begin{cases} E & \text{if } E \text{ is normal} \\ E' & \text{if } E \to^* \textbf{case } E' \textbf{ of } \ldots,\ E' \notin E \\ \text{Crits}(E') & \text{if } E \to^* \textbf{case } E' \textbf{ of } \ldots,\ E' \in E \end{cases}$$

Crits(`ord(is)`) = `is`
Crits(`srtd(ord(is))`) = Crits(`ord(is))` = `is`

Namely, we cannot evaluate `ord(is)` unless we know more about `is`.

Similarly, we cannot evaluate `srtd(ord(is))` unless we know more about `is`.

Using Critical Expressions - IND

Without Crits, following steps possible

- `IND` on `is`
- `CASE` on `ord is`
- `CASE` on `srtd(ord is)`
- `IND` on `ord is`
- `CASE` on `first(is)`
- …

….

`srtd (ordr is)`

Using Critical Expressions - IND

Apply induction on critical terms, if they are subterms of the goal and antecedents.

Apply case analysis on critical terms if they are not subterms of the goal and antecedecents.

….

```
srtd (ordr is)
```

Using Critical Expressions - IND

Apply induction on critical terms, if they are subterms of the goal and antecedents.

Crits(`srtd(ordr is)`) = { `is` }

With Crits, several steps *not* applicable

- `IND` on `is`
- ~~`CASE` on `ord is`~~
- ~~`CASE` on `srtd(ord is)`~~
- ~~`IND` on `ord is`~~
- ~~`CASE` on `first (is)`~~
- …

….

`srtd (ordr is)`

Using Critical Expressions - IND

reduces the proof search space

Crits(`srtd (ordr is)`) = { `is` }

With Crits, several steps not applicable

- `IND` on `is`
- ~~`CASE` on `ord is`~~
- ~~`CASE` on `srtd(ord is)`~~
- ~~`IND` on `ord is`~~
- ~~`CASE` on `first (is)`~~

```
 srtd (ord [])       srtd (ord js) => srtd (ord j:js)
------------------------------------------------------IND
                  srtd (ordr is)
```

Critical expressions need not be subterms

$$\text{Crits}(\texttt{E}) = \begin{cases} \texttt{E} & \text{if } \texttt{E} \text{ is normal} \\ \text{Crits}(\texttt{E}') & \text{if } \texttt{E}\texttt{->}^{*}\ \textbf{case}\ \texttt{E}'\ \textbf{of}\ \ldots,\ \ \texttt{E}' \in \texttt{E} \\ \texttt{E}' & \text{if } \texttt{E}\texttt{->}^{*}\ \textbf{case}\ \texttt{E}'\ \textbf{of}\ \ldots,\ \ \texttt{E}' \notin \texttt{E} \end{cases}$$

Critical expressions need not be subterms

$$\text{Crits}(E) = \begin{cases} E & \text{if } E \text{ is normal} \\ \text{Crits}(E') & \text{if } E \to^{*} \textbf{case } E' \textbf{ of } \ldots,\ E' \in E \\ E' & \text{if } E \to^{*} \textbf{case } E' \textbf{ of } \ldots,\ E' \notin E \end{cases}$$

```
ins i (j:js) ->* case i <= j of
                 { True -> …; False -> …}
```

Critical expressions need not be subterms

Crits(E) =	E	if E is normal
	Crits(E′)	if E->* **case** E′ **of** …, E′ ∈ E
	E′	if E->* **case** E′ **of** …, E′ ∉ E

```
ins i (j:js) ->* case i <= j of
                 { True -> …; False -> …}
```

Crits(`ins i (j:js)`) = `i<=j`

Critical expressions need not be subterms

$$
\text{Crits}(E) = \begin{cases} E & \text{if } E \text{ is normal} \\ \text{Crits}(E') & \text{if } E \to^* \textbf{case}\ E'\ \textbf{of}\ \ldots,\ E' \in E \\ E' & \text{if } E \to^* \textbf{case}\ E'\ \textbf{of}\ \ldots,\ E' \notin E \end{cases}
$$

```
ins i (j:js) ->* case i <= j of
                 { True -> …; False -> …}
```

Crits(`ins i (j:js)`) = `i<=j`

Namely, we cannot evaluate `ins i (j:js)` unless we know more about `i<=j`.

Use of critical Expressions which are not subterms are used for case analysis - 2

Apply induction on critical terms, if they are subterms of the goal and antecedents.

Apply case analysis on critical terms if they are not subterms of the goal and antecedecents.

Crits (`srtd(ins i (j:js))`) = `i<=j`

```
_______________________________??      _______________________________??

i<=j = True =>                          i<=j = False =>
   srtd (j:js)    =>                       srtd (j:js)    =>
   srtd( ins i (j:js) )                    srtd( ins i (j:js) )
_________________________________________________________________CASE
          srtd (j:js)    =>    srtd( ins i (j:js) )
```

However, consider …

```
        …                      ???
 ────────────────    ──────────────────────────────────────── ???
 srtd (ord [])        srtd (ord js) => srtd (ord j:js)
 .
 ─────────────────────────────────────────────────────────────── IND
                    srtd (ordr is)
```

However, consider …

Crits (srtd(ordr js)) = js

```
     ...                              ???
─────────────────   ─────────────────────────────────────────── ???
srtd (ord [])       srtd (ord js) => srtd (ord j:js)
.
──────────────────────────────────────────────────────────────── IND
                   srtd (ordr is)
```

However, consider …

Crits (`srtd(ordr js)`) = `js`

Should we apply induction on `js`?

```
      …                                    ???
_______________        ____________________________________________________ ???
srtd (ord [])          srtd (ord js) => srtd (ord j:js)
 .
________________________________________________________________________________ IND
                              srtd (ordr is)
```

However, consider …

Crits (`srtd(ordr js)`) = `js`

Should we apply induction on `js`?

The critical terms allow us to apply induction on `js`.

```
           …                                           ???
____________________          _________________________________________________________???
srtd (ord [])                 srtd (ord js) => srtd (ord j:js)
 .
________________________________________________________________________________________IND
                                 srtd (ordr is)
```

However, consider …

Crits (`srtd(ordr js)`) = `i<=j`

Should we apply induction on `js`?

The critical terms allow us to apply induction on `js`.

Again induction?

☹

```
         ...                                      ???
___________________         __________________________________________________???
srtd (ord [])               srtd (ord js) => srtd (ord j:js)
.__________________________________________________________________________IND
                              srtd (ordr is)
```

Zeno's trimming heuristics

- Prioritize CON and EQL steps.
- Search for counterexample after GEN steps.
- Critical expressions.
- Critical paths.
- ….

Critical Pairs

We enhance our approach so that

P1 Case statements are labeled.

Critical Pairs

We enhance our approach so that

P1 Case statements are labeled.

P2 Critical expressions are decorated with paths of labels; these describe the “intention” of the expression, ie the case statements that this expression would represent.

Critical Pairs

We enhance our approach so that

P1 Case statements are labeled.

P2 Critical expressions are decorated with paths of labels; these describe the "intention" of the expression, ie the case statements that this expression would represent.

P3 Variables are decorated with paths of labels; these describe the "history" of these variables, ie case statements that these variables have represented.

Critical Pairs

We enhance our approach so that

- P1 Case statements are labeled.
- P2 Critical expressions are decorated with paths of labels; these describe the “intention” of the expression, ie the case statements that this expression would represent.
- P3 Variables are decorated with paths of labels; these describe the “history” of these variables, ie case statements that these variables have represented.
- P4 Induction avoids revisiting (parts of) an already visited path. Therefore, induction not applicable when history of critical expression “covers” its intention.

  Similar for case analysis, generalization, etc.

P1: Labelling Case Statements - examples

…

```
letrec srtd = λ ns. case^s1 ns of
      { [] -> True;
        x:xs -> case^s2 xs of
              { [] -> True;
                y:ys -> case^s3 x<=y of
                     { True -> srtd (y:ys);
                       False -> False } }   }

letrec ordr = λ ns. case^o1 ns of
      { [] -> [];
        x:xs -> ins n (ordr xs)} }   }

letrec ins = λ n. λ ns. case^i1 ns of
      { [] -> n:[];
        x:xs -> case^i2 n<=x of
                     { True -> n:x:xs;
                       False -> x:(ins n xs)} }   }
```

P2: Decorating critical expressions - examples

```
ord(is[]) ->* case°1 is of { [] -> …; x:xs -> … }
srtd(ord(is[])) ->* cases1 ord(is) of
                    { True -> …; False -> … }
```

P2: Decorating critical expressions - examples

```
ord(is[]) ->* caseo1 is of { [] -> …; x:xs -> … }
srtd(ord(is[])) ->* cases1 ord(is) of
                         { True -> …; False -> … }
```

Crits(`ord(is[])`) = `is[],o1.[]`

When `is[]` is taken for `ord(is[])`, it “intends” to cover case **o1**

P2: Decorating critical expressions - examples

```
ord(is[]) ->* caseo1 is of { [] -> …; x:xs -> … }
srtd(ord(is[])) ->* cases1 ord(is) of
                          { True -> …; False -> … }
```

Crits(`ord(is[])`) = `is[],o1.[]`

`is[]` has not yet "covered" any cases.
If `is[]` is taken, it will cover case **o1**

Crits(`srtd(ord(is[]))`) = `is[],s1.o1.[]`

`is[]` has not yet "covered" any cases.
If `is[]` is taken, it will cover case **s1.o1**

P3: Decorating variables

-

$$\frac{\qquad}{\texttt{srtd (ordr is}^{\texttt{[]}}\texttt{)}}\ \texttt{IND}$$

P4: Induction – only when intention is not "covered" by history

x has type T, **x^p, p' ∈ Crits (ϕ)**
…x^p''… , p''' ∈ Crits (ϕ) implies p' not a sub-path of p''
for each K ∈ Constrs (T) . ⊢ ϕ [x:=z$_1$], … ϕ [x:=z$_m$] => ϕ [x:=K y$_1$…y$_n$]
where …

——————————————————————————— IND

⊢ ϕ

P4: Induction – only when intention is not "covered" by history

Crits (`srtd(ordr is[])`) = `is[]`,**`p1`**

where
`p1 = s1.o1.[]`

Therefore, IND applicable now. ☺

x has type `T`, **x**^{p}, **p'** ∈**Crits** (ϕ)
…x$^{p''}$**… , p''' ∈Crits (ϕ) implies p' not sub-path of p''**
for each `K` ∈Constrs (`T`) . ⊢ ϕ[x:=z_1], … ϕ[x:=z_m] => ϕ[x:=K y_1…y_n]
where …
——————————————————————————— IND
⊢ ϕ

.
——————————————————————————— IND
`srtd (ordr is[])`

Second step in proof

Remember, here we wanted to avoid application of induction.

```
                                                            ???
_______…__________          _____________________________________________________???
srtd (ord [])          srtd (ord js^p1) => srtd (ord j^p1:js^p1)
 .______________________________________________________________________________IND
                              srtd (ordr is^[])
```

P4: Induction only applicable when intention not covered by history

Crits (srtd(ordr js^{p1})) = js^{p1},**p1**

Crits (srtd(ordr j^{p1}:js^{p1})) = js^{p1},**p1**

where

p1 = s1.o1.[]

…

srtd (ord [])

???

srtd (ord js^{p1}) => srtd (ord j^{p1}:js^{p1}) ???

.

srtd (ordr is$^{[]}$) IND

P4: Induction only applicable when intention not covered by history

Crits (`srtd(ordr js`^{p1}`))` = `js`^{p1}, **p1**

Crits (`srtd(ordr j`^{p1}`:js`^{p1}`))` = `js`^{p1}, **p1**

where

p1 = s1.o1.[]

Therefore, IND not applicable now. ☺

$$\frac{\begin{array}{l}x \text{ has type } T,\ x^{p},\ p' \in \text{Crits}(\phi)\\ \ldots x^{p''}\ldots,\ p''' \in \text{Crits}(\phi) \text{ implies } p' \text{ not a subpath of } p''\\ \text{for each } K \in \text{Constrs}(T).\ \vdash \phi[x{:=}z_1], \ldots \phi[x{:=}z_m] \Rightarrow \phi[x{:=}K\ y_1\ldots y_n]\\ \quad \text{where} \ldots\end{array}}{\vdash \phi}\ \text{IND}$$

$$\frac{\dfrac{\ldots}{\texttt{srtd (ord [])}} \qquad \dfrac{???}{\texttt{srtd (ord js}^{p1}\texttt{) => srtd (ord j}^{p1}\texttt{:js}^{p1}\texttt{)}}\ ???}{\texttt{srtd (ordr is)}}\ \text{IND}$$

Summary

- Zeno proves equality over Haskell-like terms.
- Variables implicitly universally quantified; no support for existentials. Booleans are encoded through the `Bool` data type.
- From Isaplanner benchmark suite, Zeno can prove more properties than Isaplanner and ACL2s
- Zeno often discovers useful further lemmas.
- Zeno's heuristics
  - Counteraxamples
  - Prioritize EQL and CON
  - Critical expressions restrict antecedents to "relevant ones" - they move the proof search towards making it possible to expand function bodies – as opposed to rippling
  - Paths keep track of the proof cases visited so far and avoid revisiting these cases; some "forbidden" steps my become allowed later in the poof.
  - …

To Do - s

- Formal Underpinnings
- Expand Zeno
- Adapt Zeno to handle “families of proofs”
- Adapt Zeno for Program Verification

Zeno

Contents [hide]

1 Introduction

Zeno is an automated proof system for Haskell program properties; developed at Imperial College London by William Sonnex, Sophia Drossopoulou and Susan Eisenbach. It aims to solve the general problem of equality between two Haskell terms, for any input value.

Many program verification tools available today are of the model checking variety; able to traverse a very large but finite search space very quickly. These are well suited to problems with a large description, but no recursive datatypes. Zeno on the other hand is designed to inductively prove properties over an infinite search space, but only those with a small and simple specification.