19 Towards Program verification

19.1 Integer theorem proving in PRL

The formalization of Arithmetic used in the PRL system aims at a hybrid solution to the problem of non-axiomatizability. We formalize the full integer arithmetic using Peano arithmetic with induction schemata and many predefined operations and predicates. As one of the proof rules we provide a decision procedure for the sub-theory of elementary arithmetic, which helps us to decide the most fundamental arithmetical statements.
We're talking about integer arithmetic with the standard domain $\mathbb{Z}$ and use in the formal language, besides first-order logic with equality the following symbols:

- Integer constants in decimal representation: `0, 1, -1, 2, -2, ...`
- The unary function symbol `-` (Negation) in prefix notation

  The binary function symbols `+`, `-`, `*`, $\div$, `rem` in infix notation.
  (addition, subtraction, multiplication, division, quotient remainder)
- A constructor for inductively defined functions $\texttt{ind}(i,y.t_d[i,y];\ t_b;\ i,y.t_u[i,y])$

  which defines a function f by primitive recursion both for positive and negative numbers. A more convenient way of writing is

  $$\begin{array}{lll} f \texttt{ with } f(i) = & t_d[i,y] & \texttt{if } i\texttt{<0, where } y\texttt{=}f(i\texttt{+1}), \\ & t_b & \texttt{if } i\texttt{=0} \qquad , \\ & t_d[i,y] & \texttt{if } i\texttt{>0, where } y\texttt{=}f(i\texttt{-1}). \end{array}$$

  Note that the induction goes from n-1 to n in the positive case instead of from i to i+1. This allows a more uniform description of the value f(i).

  With this operator we can, for instance, define exponentiation x^i as

  $$\begin{array}{lll} f \texttt{ with } f(i) = & \texttt{0} & \texttt{if } i\texttt{<0, where } y\texttt{=}f(i\texttt{+1}), \\ & \texttt{1} & \texttt{if } i\texttt{=0} \qquad , \\ & \texttt{x*y} & \texttt{if } i\texttt{>0, where } y\texttt{=}f(i\texttt{-1}). \end{array}$$
- The binary predicate symbol `<` and, derived from it `>`, $\leq$, $\geq$ in infix notation

Our axioms are, besides the rules of first-order logic with equality, the following adapted axioms from Peano Arithmetic $\boxed{\textit{see handout}}$

1. The axioms of *constant arithmetic*: `1+1=2, 2+1=3, 3+1=4, ...`
2. Evaluation axioms for inductively defined functions

   $f \texttt{ = ind}(i,y.t_d[i,y];\ t_b;\ i,y.t_u[i,y])$ implies

   $i\texttt{<0} \Rightarrow f(i) = t_d[i, f(i{+}1)] \quad \wedge \quad f(0) = t_b \quad \wedge \quad i\texttt{>0} \Rightarrow f(i) = t_u[i, f(i{-}1)]$

   This axiom can also be expressed by a set of rule schemes

   $\Gamma,\ i\texttt{>0} \vdash P[f(i)]$ `BY eval` f
   $\quad\Gamma,\ i\texttt{>0} \vdash P[t_u[i, f(i{-}1)]]$

   $\Gamma,\ i\texttt{<0} \vdash P[f(i)]$ `BY eval` f
   $\quad\Gamma,\ i\texttt{<0} \vdash P[t_d[i, f(i{+}1)]]$

   $\Gamma \vdash P[f(0)]$ `BY eval` f
   $\quad\Gamma \vdash P[t_b]$

   where f is inductively defined as above.

3. The *ring axioms* of the integers (Commutativity, Associativity, Distributivity, Identities, Additive inverse, Definition of subtraction)
4. Axioms of *discrete linear order* for `<` and axioms defining the other inequalities
5. *Monotonicity* axioms of `+,-,*`

   The definition or $\div$ and rem: $\texttt{x}\div\texttt{y=q} \wedge \texttt{x rem y=r} \quad\Leftrightarrow\quad \texttt{x = q*y+r} \wedge \texttt{0}\leq\texttt{r} \wedge \texttt{r<y}$
6. The *induction scheme* for integers: *for every formula P with free variable z*

   $P[0/z] \wedge (\forall i)(i{>}0 \Rightarrow P[i{-}1/z] \Rightarrow P[i/z]) \wedge (\forall i)(i{<}0 \Rightarrow P[i{+}1/z] \Rightarrow P[i/z]) \;\Rightarrow\; (\forall x)(P[x/z])$

   The induction axiom can also be expressed as rule scheme

   $i:\mathbb{Z} \vdash P[i/z]$ by induction on i using $z.P[z]$
   1. $i:\mathbb{Z}$, i<0, $P[i{-}1/z] \vdash P[i/z]$
   2. $\vdash P[0/z]$
   3. $i:\mathbb{Z}$, i>0, $P[i{+}1/z] \vdash P[i/z]$

   We sometimes also use a simpler derived rule scheme when we only talk about natural numbers:

   $n:\mathbb{Z}$, $n{\geq}0 \vdash P[n/z]$ by NatInd
   1. $\vdash P[0/z]$
   2. $n:\mathbb{Z}$, n≥0, $P[n/z] \vdash P[n{+}1/z]$

   I usually abbreviate $n:\mathbb{Z}$, $n{\geq}0$ by $n:\mathbb{N}$ to make this look simpler

Now it would be extremely tedious to prove anything formally with just these axioms because you need extremely many steps to get to any meaningful result. So if you want to use formal systems for anything only remotely related to practical applications, you need a stronger rule. The PRL system therefore not only provides an expressive formalization of arithmetic but also a decision procedure for the subset of elementary arithmetic, which consists of virtually all the trivial arithmetical insights that pop up so often during a proof. This rule, called `Arith` in refinement logic can be called as follows:

$$H_1, \ldots H_m \vdash G_1 \vee \ldots \vee G_n \quad \text{by Arith}$$

and it is used to automatically solve problems like

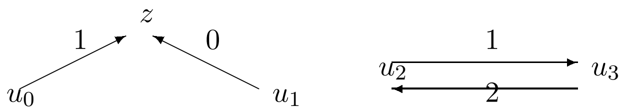
- `0<x` $\vdash$ `0<x+2`
- Γ, x≤y, x≠y $\vdash$ x<y
- Γ, x+y>z, 2*x≥z, x+y+2*x≥z+z+1 $\vdash$ 3*x+y≥2*z-1

So I will briefly discuss how this rule operates, the theory it can decide and where its limitations are. Most informations you will find on the HANDOUT.

1. The theory $\mathcal{A}$ of elementary arithmetic
2. The `Arith` rule applies to an arbitrary proof goal but ignores all H_i and G_j that cannot be understood as atomic formula in $\mathcal{A}$. It completes the (partial) proof if $G \vee \ldots \vee G_n$ follows from the hypotheses H_1, ... H_m in the theory $\mathcal{A}$. Otherwise the procedure `Arith` fails.
3. The decision procedure essentially works as follows – Example:
   (a) Γ, x+y>z, 2*x≥z, x+y+2*x≥z+z+1, ¬(3*x+y≥2*z-1) $\vdash$ false
   (b) —
   (c) —
   (d) Γ disappears: x+y>z, 2*x≥z, x+y+2*x≥z+z+1, ¬(3*x+y≥2*z-1) $\vdash$ false

(e) $u_0 > z,\ u_1 \geq z,\ u_2 \geq 1 + u_3,\ \neg(u_2 \geq (-1) + u_3) \ \vdash\ \texttt{false}$
with $u_0 \equiv x+y,\ u_1 \equiv 2*x,\ u_2 \equiv 3*x+y$ and $u_3 \equiv 2*z$

(f) $u_0 \geq z+1,\ u_1 \geq z,\ u_2 \geq 1+u_3,\ u_3 \geq u_2+2 \ \vdash\ \texttt{false}$
Conversions are $\neg x \geq y \Leftrightarrow x<y,\quad x>y \Leftrightarrow x \geq y+1,$
$x=y \Leftrightarrow x \geq y \wedge y \geq x,\quad x \neq y \Leftrightarrow x \geq y+1 \vee y \geq x+1,$ *as well as adding constants*

(g) 

(h) Standard path-checking algorithm says there is a positive cycle thus the formula set is contradictory, i.e. the original goal is valid.

4. `Arith` handles only *Trivial monotonicities*, i.e. instances of the monotonicity laws of '+' and '-' where `z` and `w` are constants. The more complex monotonicities need interaction. So goals like `0<x ⊢ 0<x*x` or `Γ, x+y>z, 2*x≥z ⊢ 3*x+y ≥ 2*z-1`, however, require explicit applications of a `monotonicity` rule before calling `Arith`.

Now that we have `Arith`, we are able to prove interesting theorems, for instance that every natural number has an integer square root. I use x:IN as abbreviation for x:$\mathbb{Z}$, $x \geq 0$:

```
⊢ ∀n:IN.∃y:IN. y²≤x ∧ x<(y+1)²                                         BY allR
\
 n:IN ⊢ ∃y:IN. y²≤x ∧ x<(y+1)²                                         BY NatInd
  |\
  | ⊢ ∃y:IN. y²≤0 ∧ 0<(y+1)²                                           BY exR 0
  |   \
  |    ⊢ 0²≤0 ∧ 0(0+1)²                                                BY andR
  |     |\
  |     |  ⊢ 0²≤0                                                      BY Arith
  |      \
  |        ⊢ 0(0+1)²                                                   BY Arith
   \
    n:IN, ∃y:IN. y²≤n ∧ n<(y+1)² ⊢ ∃y:IN. y²≤n+1 ∧ n+1<(y+1)²          BY exL 2 THEN andL 3
     \
      n,y:IN, y²≤n, n<(y+1)² ⊢ ∃y:IN. y²≤n+1 ∧ n+1<(y+1)²              BY cut  n<(y+1)² ∨ n=(y+1)²
      |\
      | n,y:IN,  y²≤n, n<(y+1)² ⊢ n+1<(y+1)² ∨ n+1=(y+1)²              BY Arith
       \
        n,y:IN, y²≤n, n<(y+1)², n+1<(y+1)² ∨ n+1=(y+1)² ⊢ ∃y:IN. y²≤n+1 ∧ n+1<(y+1)² BY orL 5
        |\
        | n,y:IN, y²≤n, n<(y+1)², n+1<(y+1)² ⊢ ∃y:IN. y²≤n+1 ∧ n+1<(y+1)²   BY exR y
        |  \
        |   n,y:IN, y²≤n, n<(y+1)², n+1<(y+1)² ⊢  y²≤n+1 ∧ n+1<(y+1)²      BY andR
        |   |\
        |   | n,y:IN, y²≤n, n<(y+1)², n+1<(y+1)² ⊢  y²≤n+1                 BY Arith
        |    \
        |     n,y:IN, y²≤n, n<(y+1)², n+1<(y+1)² ⊢  n+1<(y+1)²             BY hypothesis 6
         \
          n,y:IN, y²≤n, n<(y+1)², n+1=(y+1)² ⊢ ∃y:IN. y²≤n+1 ∧ n+1<(y+1)²   BY exR y+1
           \
            n,y:IN, y²≤n, n<(y+1)², n+1=(y+1)² ⊢ (y+1)²≤n ∧ n<(y+1+1)²     BY andR
            |\
            | n,y:IN, y²≤n, n<(y+1)², n+1=(y+1)² ⊢ (y+1)²≤n+1              BY Arith
             \
              n,y:IN, y²≤n, n<(y+1)², n+1=(y+1)² ⊢ n+1<(y+1+1)²            BY Arith
```

Now this proof has an interesting *computational meaning*. It not only says THAT every natural number has an integer square root but also describes how to construct it. We proceed inductively, assign 0 in the base case, make a case disctinction in the step case and use either either the previous value or increase it by one, if we reach a square number. Given the right mechanisms, we would be able to extract a program from this proof. In principle, this holds for all ∀∃-theorems. We therefore call this class of theorems also *formal specifications* of programs.

OPTIONAL HOMEWORK:

1. Using the constructor for inductively defined functions show how to express a the case distinctions `if` i<j `then` a `else` b and `if` i=j `then` a `else` b where i, j are arithmetical expressions and a, b are arbitary expressions.

   `if` i<j `then` a `else` b $\equiv$ $f(i$-$j)$ *where f abbreviates*

   f `with` $f(i)$ = a `if` i<0, `where` y=$f(i$+1$)$, b `if` i=0, b `if` i>0, `where` y=$f(i$-1$)$.

   `if` i=j `then` a `else` b $\equiv$ $g(i$-$j)$ *where f abbreviates*

   g `with` $g(i)$ = b `if` i<0, `where` y=$g(i$+1$)$, a `if` i=0, b `if` i>0, `where` y=$g(i$-1$)$.

2. Using case distinction and the constructor for inductively defined functions define a function $\sqrt{\ }$ that computes the integer square root of a natural number n. Try to make it match the formal proof for ∀n:IN.∃y:IN. y²≤x ∧ x<(y+1)². *Optional: formally prove it correct*

```
f with f(i) = 0 if i<0, where y=f(i+1),
              0   if i=0                ,
              if i<(y+1)² then y else y+1  if i>0, where y=f(i-1).
```

   Correctness proof is:

```
⊢ ∀n:IN. √n²≤x ∧ x<(√n+1)²                                        BY allR
\
 n:IN ⊢  √n²≤x ∧ x<(√n+1)²                                         BY NatInd
 |\
 | ⊢  √0²≤0 ∧ 0<(√0+1)²                                            BY eval √0
 |  \
 |   ⊢ 0²≤0 ∧ 0(0+1)²                                               BY andR
 |   |\
 |   |  ⊢ 0²≤0                                                      BY Arith
 |    \
 |      ⊢ 0(0+1)²                                                   BY Arith
  \
   n:IN, √n²≤n ∧ n<(√n+1)² ⊢  √n²≤n+1 ∧ n+1<(√n+1)²                BY andL 2
    \
     n:IN, √n²≤n, n<(√n+1)² ⊢  √n²≤n+1 ∧ n+1<(√n+1)²               BY cut  n<(√n+1)² ∨ n=(√n+1)²
     |\
     | n:IN,  √n²≤n, n<(√n+1)² ⊢ n+1<(√n+1)² ∨ n+1=(√n+1)²         BY Arith
      \
       n:IN, √n²≤n, n<(√n+1)², n+1<(√n+1)² ∨ n+1=(√n+1)² ⊢  √n+1²≤n+1 ∧ n+1<(√n+1+1)² BY orL 4
       |\
       | n:IN, √n²≤n, n<(√n+1)², n+1<(√n+1)² ⊢  √n+1²≤n+1 ∧ n+1<(√n+1+1)²   BY eval √n+1
       |  \
       |   n:IN, √n²≤n, n<(√n+1)², n+1<(√n+1)² ⊢  √n²≤n+1 ∧ n+1<(√n+1)²     BY andR
       |   |\
       |   | n:IN, √n²≤n, n<(√n+1)², n+1<(√n+1)² ⊢  √n²≤n+1                 BY Arith
       |    \
       |     n:IN, √n²≤n, n<(√n+1)², n+1<(√n+1)² ⊢  n+1<(√n+1)²              BY hypothesis 6
        \
         n:IN, √n²≤n, n<(√n+1)², n+1=(√n+1)² ⊢  √n+1²≤n+1 ∧ n+1<(√n+1+1)²   BY  eval √n+1
          \
           n:IN, √n²≤n, n<(√n+1)², n+1=(√n+1)² ⊢ (√n+1)²≤n ∧ n<(√n+1+1)²    BY andR
           |\
           | n:IN, √n²≤n, n<(√n+1)², n+1=(√n+1)² ⊢ (√n+1)²≤n+1              BY Arith
            \
             n:IN, √n²≤n, n<(√n+1)², n+1=(√n+1)² ⊢ n+1<(√n+1+1)²            BY Arith
```

3. Formalize the statement that any natural number greater than or equal to 8 can be expressed as the sum of some number of 3's and 5's (e.g. $11 = 3+3+5$). Prove it formally and discuss any possible computational meaning of your proof.

By the way: how many of you had problems with assignment #4 for today?
Would you think you can solve it if you have to turn it in in 2 weeks?

19.2 Lists (λ-PRL)

Now that we have arithmetic and seen how to specify programs by theorems, we're already pretty close to using formal logic for the analysis of programs. To allow the study of more realistic programs we need to introduce data structures.
In principle, it is possible to encode all denumerable domains by numbers and to perform the computations on the numbers representing the data we are interested in. So to represent a function $f : D_1 \rightarrow D_2$ where D_1 and D_2 are arbitrary denumerable domains, we assume that we have surjective mappings $\delta_1 : \mathbb{N} \rightarrow D_1$ and $\delta_2 : \mathbb{N} \rightarrow D_2$ and then represent f by a function $\hat{f} : \mathbb{N} \rightarrow \mathbb{N}$ with the property $f(\delta_1(n)) = \delta_2(\hat{f}(n))$ for all $n \in \mathbb{N}$, or briefly $f \circ \delta_1 = \delta_2 \circ \hat{f}$ $\boxed{\text{Draw Diagram}}$.
For the representation of lists, one of the most important data structures, we could use the encoding with prime numbers that I mentioned last tuesday.

> To encode a sequence $s_1, s_2, .., s_n$ of natural numbers we
>
> 1. search for the smallest prime number p that is larger than all the s_i
> 2. represent $s_1, s_2, .., s_n$ by the number $\hat{s} = s_0 + s_1 * p + s_2 * p^2 \, .. + s_n * p^n$
>
> This is a very old encoding from the time where one didn't care much about efficiency but only on a simple proof that it is actually an encoding that can be decoded. Mathematicians were quite fond of primes at the time.

Here is another one, that is very popular in computer science.
A simple way to enumerate all pairs of natural numbers in a bijective way proceeds as follows.

	0	1	2	3	4	5	...
0	0	2	5	9	14	20	...
1	1	4	8	13	19		...
2	3	7	12	18			...
3	6	11	17				...
4	10	16					...
5	15						...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	...

So the encoding works as follows: $\langle x, 0\rangle = \sum_{i=0}^{x} i = x * (x+1)/2$ and $\langle x, y\rangle = \langle x+y, 0\rangle + y$.
After a few consideration one discovers that we can describe this standard tupling function as $\langle x, y\rangle := (x+y)(x+y+1)/2 + y$. This function is bijective and primitive-recursive. Its inverses are easy to compute as well as you can see by looking at the table. So it is good for computational purposes.
Now you can extend this function to a bijective enumeration of k-tuples of natural numbers by $\langle\rangle^{(1)}(x) := x$ and $\langle\rangle^{(k+1)}(x_1, \ldots, x_{k+1}) := \langle\langle\rangle^{(k)}(x_1, \ldots, x_k),\, x_{k+1}\rangle$

This function, which we briefly denote by $\langle x_1,\ldots,x_{k+1}\rangle$ is still bijective and easy to compute for each fixed k. Now from here, it is an easy step to represent arbitary lists: we simply encode a list $s_1, s_2, .., s_n$ by first giving its length n and then encoding the corresponding n-tuple: $\langle n,\ \langle s_1,\ldots,s_n\rangle\rangle$. This is still a bijection and easy to encode and decode. Give an number i we first determine the length n and the number j that encodes $\langle s_1,\ldots,s_n\rangle$. We then take the decoding for n-tubles and can access each individual element of the list.
Once we have tupling functions, we can actually represent integers as pairs of signs (odd = positve, even = negative) and absolute value. Rationals can be encoded as fractions of integers and non-zero natural numbers, and so on.
But when we start proving things about these programs on these data, working with the numbers representing certain data becomes quite awkward, particularly because we won't always have a good bijective representation.

So it is more reasonable to extend first-order logic with equality and arithmetic by the basic axioms of the data types we are interested in.

I will focus on one basic data structure, *lists of integers*, because you can do quite a lot once you have these. We add the following to our formal theory of integer arithmetic.

- The *formal language* characterizes lists of integers

  Constant lists: `[]`, `[`i_1`;`i_2`; ...` i_n`]` where the i_j are integer terms

  The *binary function symbol* `::` in infix notation.
  (prepending an element to a list – we say "cons")

  A constructor for inductively defined list functions `list_ind(`t_b`;` $i,l',y.t_u[i,l',y]$`)`

  which is also written as

  f `with` $f(l)$ `=` t_b `if` l`=[],` $t_u[i,l',y]$ `if` l`=`i`::`l' `where` y`=`$f(l')$`.`

- The axioms can be viewed as extension of Peano Arithmetic, because the length of a list characterizes natural numbers, so a constant list expands the concept of a natural number constant, and prepending corresponds to the successor function. Because of the induction operator we don't need any more predefined operations, because they all can be expressed with this operator. Here are a few examples

  Appending a list l_1 `@` l_2 as function $l_1 \mapsto l_1$`@`l_2

  f `with` $f(l_1)$ `=` l_2 `if` l_1`=[],` i`::`y `if` l_1`=`i`::`l_1' `where` y`=`$f(l_1')$`.`

  The length of a list `as function` $l \mapsto$ `|`l`|`

  f `with` $f(l)$ `=` `0` `if` l`=[],` y`+1` `if` l`=`i`::`l' `where` y`=`$f(l')$`.`

  So we only need the very basic axioms

  1. $i{::}l \neq []$
  2. $i_1{::}l_1 = i_2{::}l_2 \Rightarrow i_1 = i_2 \wedge l_1 = l_2$
  3. $P[[]/z] \wedge (\forall l,i)(P[l/z] \Rightarrow P[i :: l/z]) \Rightarrow (\forall x)(P[x/z])$ *for every formula P with free variable z*
  4. $f =$ `list_ind(`t_b`;` $i,l',y.t_u[i,l',y]$`)` implies $f([]) = t_b \wedge f(i{::}l) = t_u[i,l,f(l)]$

  Again we could express both the induction axiom scheme and the list_ind-evaluation by a set of proof rules `ListInd` and `Eval`

This theory obviously has a model, because the standard interpretation using integer lists satisfies the axioms.

With this theory, you can investigate quite a few interesting programming problems. Let us state, for instance, that every list has a maximal element.

For this purpose we need to define what it means to be an element of a list:

$\texttt{On}(x,l)$ is defined as $f(l)$ where f abbreviates

f with $f(l)$ = false if l=[],
if i=x then true else y if l=i::l' where y=$f(l')$.

Then we formulate (∀l:List)(∃i:ℤ)(On(i,l) ∧ (∀j:ℤ)(On(j,l) ⇒ j≤i))

What is wrong with this statement? We can't prove it because empty lists don't have elements. So let us refine the theorem into

(∀l:List)(l≠[] ⇒ (∃i:ℤ)(On(i,l) ∧ (∀j:ℤ)(On(j,l) ⇒ j≤i)))

A proof of this theorem would tell us how to compute the maximum of a nonempty list. So it specifies a very valuable function that we can, for instance, use for sorting lists. The proof is already quite messy, because we don't have a decision procedure for lists. Here is a sketch

```
⊢ (∀l:List)(l≠[] ⇒ (∃i:ℤ)(On(i,l) ∧ (∀j:ℤ)(On(j,l) ⇒j≤i)))  BY allR
 l:List ⊢ l≠[] ⇒ (∃i:ℤ)(On(i,l) ∧ (∀j:ℤ)(On(j,l) ⇒j≤i)))  BY ListInd

  ⊢ []≠[] ⇒ (∃i:ℤ)(On(i,[]) ∧ (∀j:ℤ)(On(j,[]) ⇒j≤i)))  BY impR
   ⊢ []≠[] ⊢  (∃i:ℤ)(On(i,[]) ∧ (∀j:ℤ)(On(j,[]) ⇒j≤i)))  BY notL
    ⊢ []≠[] ⊢ []=[] by reflexivity

l:List, l≠[] ⇒ ...., k:ℤ ⊢ k::l≠[] ⇒ (∃i:ℤ)(On(i,k::l) ∧ (∀j:ℤ)(On(j,k::l) ⇒j≤i))) BY impR
 l:List, l≠[] ⇒ ...., k:ℤ, k::l≠[] ⊢ (∃i:ℤ)(On(i,k::l) ∧ (∀j:ℤ)(On(j,k::l) ⇒j≤i))) BY cut l=[]
∨ l≠[]

  l:List, l≠[] ⇒ ...., ... ⊢ l=[] ∨ l≠[] BY magic

  l:List, l≠[] ⇒ ...., ..., l=[] ∨ l≠[]  ⊢ (∃i:ℤ)(On(i,k::l) ∧ (∀j:ℤ)(On(j,k::l) ⇒j≤i))) BY orL
   l:List, l≠[] ⇒ ...., ..., l=[] ⊢ (∃i:ℤ)(On(i,k::l) ∧ (∀j:ℤ)(On(j,k::l) ⇒j≤i))) BY exR k
    l:List, l≠[] ⇒ ...., ..., l=[] ⊢ (On(k,k::l) ∧ (∀j:ℤ)(On(j,k::l)  ⇒j≤k))) BY  messy details

   l:List, l≠[] ⇒ ...., ..., l≠[] ⊢ (∃i:ℤ)(On(i,k::l) ∧ (∀j:ℤ)(On(j,k::l) ⇒j≤i))) BY impL
    l:List, (∃i:ℤ)(On(i,l) ∧ (∀j:ℤ)(On(j,l) ⇒j≤i))), ..., l≠[]
         ⊢ (∃i:ℤ)(On(i,k::l) ∧ (∀j:ℤ)(On(j,k::l) ⇒j≤i))) BY exL THEN andL
     l:List, i:ℤ, On(i,l), (∀j:ℤ)(On(j,l) ⇒j≤i))), ..., ...
         ⊢ (∃i:ℤ)(On(i,k::l) ∧ (∀j:ℤ)(On(j,k::l) ⇒j≤i))) BY cut i<k ∨i≥k
    ⋮
```

(I don't need magic, it's just more convenient here)

HOMEWORK:

1. Using the available constants and functions formally define the following concepts
   (a) increasing(l), saying that l is an increasing list
   (b) index(l,i), computing the i-th element of l (for $1 \leq i \leq |l|$)
   (c) multiplicity(x,l), the number of occurrences of x in l (0 to $|l|$ possible)

   With these concepts state the theorem that any list can be sorted into an increasing order.

   What would be the computational meaning of a proof of that theorem?

Opt: Specify formally that any list has a reversal (e.g. the reversal of [1;2;7;3;1] is [1;3;7;2;1])