25 λ-PRL

λ-PRL (**p***roof* **r***efinement* **l***ogic, version lambda*) is a logic for reasoning about programs, integers and lists of integers. These concepts are not formalized in first-order logic and characterized through axioms but explicitly part of the formal language. Besides first-order logic with equality and strict order, it uses integer constants, lists constants, function symbols, a few predefined functions – some of which using infix notation, and terms for inductively defined functions.[1]

25.1 The formal language

- **Types**: `int` and `list`
- **Variables**: $a, b, c, x, y, \ldots$.
- **Function symbols**: $f, g, h, \ldots$ — **will be associated with types**
- **Predicate symbols**: $P, Q, R, \ldots$ — **will be associated with types**
- **Constants**:

  Integer constants in decimal representation: `0`, `1`, `-1`, `2`, `-2`, ...

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

  Constant function symbols: — **associated with types**
  – The unary function symbol `-` — `int`×`list` → `list`
  (Negation)
  – The binary operators `+`, `-`, `*`, `/`, `mod` — `int`×`int` → `int`
  (addition, subtraction, multiplication, division, quotient remainder)
  – The binary operator `::` — `int`×`list` → `list`
  (prepending an element to a list)

  Constant predicate symbols: — **associated with types**
  – The binary predicate symbols `<` and `=`
  (Predicate symbols `>`, $\leq$, $\geq$ *can be defined through language extension)*

- **Terms** are:
  – constants and variables
  – function applications $f(t_1, ..t_n)$ — terms t_i must match the types of f
  – infix operator applications t_1 *op* t_2 — terms t_i must match the types of *op*
  – listings $[t_1, ..t_n]$ — terms t_i must have type `int`

- **Formulas** are:

  Atomic formulas:
  – t_1 `=` t_2 — terms t_i must have the same type
  – $t_1 < t_2$ — terms t_i must have type `int`

  Compound formulas:
  – $\neg A$, $A \wedge B$, $A \vee B$, $A \Rightarrow B$, $\forall x_1, ..x_n{:}T.\ A$, $\exists x_1, ..x_n{:}T.\ A$ — T is `int` or `list`

[1] Some of the formalities differ slightly from the λ-PRL manual to be consistent with the notation used in class. Note that the handouts on April 24 contained several mistakes due to careless editing (CK).

25.2 Language Extensions

The formal language of λ-PRL may be extended by user-defined concepts through *definitions*:

$$\textit{template with formal parameters} \equiv \textit{expression}$$

For instance

$$\begin{array}{lcl} x\texttt{≤}y & \equiv & x\texttt{=}y \ \vee \ x\texttt{<}y \\ \exists\texttt{!}x\texttt{:}T\texttt{.}\ P[x] & \equiv & \exists x\texttt{:}T\texttt{.}\ P[x] \ \wedge \ \forall\texttt{y:}T\texttt{.}\ P[\texttt{y}/x] \ \Rightarrow \ x\texttt{=y} \end{array}$$

Note that expressions in math-font (x, T, $P[x]$) are formal parameters of the template and the expression, while `y` is not. The notation $P[x]$ means that P is an expression with possible free occurrences of the variable x. $P[t/x]$ denotes that every free occurrence of x is replaced by the term t.

In λ-PRL, definitions are like text macros: formal parameters on the left hand side cause the corresponding term slots on the right hand side to be instantiated (the mechanism for terms with free variables is more sophisticated). So `4`≤`5` will be expanded to `4=5`∨`4<5` and ∃`!x:int.(x<5`∧`3<x)` to ∃`x:int.(x<5`∧`3<x)` ∧ ∀`y:int.(y<5`∧`3<y)` ⇒ `x=y`.

The expression on the right hand side of a definition can be any term and formula. In addition to that we λ-PRL supports *inductively defined function*, built with one of the following two constructs.

- Inductive definition on integers, using primitive recursion both for positive and negative numbers. The general form is

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

  The terms t_b, t_d, and t_u denote the base case of the inductive definition as well as the down- and upgoing recursions. The latter may depend on the input value i and the result y of the recursive predecessor, i.e. f`(`i`+1)` for induction on negative numbers and f`(`i`-1)` for induction on positive numbers.

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

- Inductive definition on lists, using the form

  $$\begin{array}{lcll} f\texttt{(}l\texttt{)} & \equiv & t_b & \texttt{if } l\texttt{=[]} \\ & & t_u[i,tl,y] & \texttt{if } l\texttt{=}i\texttt{::}tl \texttt{ where } y\texttt{=}f\texttt{(}tl\texttt{)} \end{array}$$

  Here the value t_u for the step case may depend on the input value l, which consists of a first element i and a tail tl, and the result y of the recursive predecessor f`(`tl`)`

With this operator we can, for instance, define exponentiation x^i as a function `exp`

$$\begin{array}{lcll} \texttt{exp(}x\texttt{,}i\texttt{)} & \equiv & \texttt{0} & \texttt{if } i\texttt{<0, where y=exp(}x\texttt{,}i\texttt{+1)} \\ & & \texttt{1} & \texttt{if } i\texttt{=0} \\ & & x\texttt{*y} & \texttt{if } i\texttt{>0, where y=exp(}x\texttt{,}i\texttt{-1)} \end{array}$$

A note on differences between λ-PRL and NuPRL

The description of recursive definitions presented in class on April 24, was taken from NuPRL instead of λ-PRL. In NuPRL a recursive definition is a part of the formal language. There is a *term* for inductively defined concepts on integers, and the above definition could be expressed as

$f(x) \equiv \mathsf{ind}(x;\ i,y.t_d[i,y];\ t_b;\ i,y.t_u[i,y])$

where the right hand side of that definition is usually displayed as

$$\mathtt{h}(x)\ \mathtt{with}\ \mathtt{h}(i) = \begin{array}{ll} t_d[i,y] & \mathtt{if}\ i\mathtt{<0,\ where}\ y\mathtt{=h(}i\mathtt{+1)} \\ t_b & \mathtt{if}\ i\mathtt{=0} \\ t_d[i,y] & \mathtt{if}\ i\mathtt{>0,\ where}\ y\mathtt{=h(}i\mathtt{-1)} \end{array}$$

In the same way the formal language of NuPRL provides a term for inductively defined concepts on lists.

In λ-PRL recursion requires a specialized definition mechanism, which is simpler as long as the formal language contains only a few constructs, but limits formal reasoning about inductively defined functions in more expressive languages.

25.3 Proof Rules (refinement logic)

The presence of types in the formal language makes it necessary to include *type declarations* in the list of hypotheses. $x{:}T$ denotes that x is declared to be of type T. This modification affects the quantifier rules `allR` and `exL` and the rules for induction.

1. **(Typed) predicate logic rules**:

left		right	
`andL` i	$H,\ A \wedge B,\ H' \vdash G$ $\quad H,\ A,\ B,\ H' \vdash G$	$H \vdash A \wedge B$ $\quad H \vdash A$ $\quad H \vdash B$	`andR`
`orL` i	$H,\ A \vee B,\ H' \vdash G$ $\quad H,\ A,\ H' \vdash G$ $\quad H,\ B,\ H' \vdash G$	$H \vdash A \vee B$ $\quad H \vdash A$	`orR1`
		$H \vdash A \vee B$ $\quad H \vdash B$	`orR2`
`impL` i	$H,\ A \Rightarrow B,\ H' \vdash G$ $\quad H,\ A \Rightarrow B,\ H' \vdash A$ $\quad H,\ H',\ B \vdash G$	$H \vdash A \Rightarrow B$ $\quad H,\ A \vdash B$	`impR`
`notL` i	$H,\ \neg A,\ H' \vdash G$ $\quad H,\ \neg A,\ H' \vdash A$	$H \vdash \neg A$ $\quad H,\ A \vdash$ `false`	`notR`
`allL` i t	$H,\ \forall x{:}T.\ A,\ H' \vdash G$ $\quad H,\ \forall x{:}T.\ A,\ A[t/x],\ H' \vdash G$	$H \vdash \forall x{:}T.\ A$ $\quad H,\ x{:}T \vdash A$	`allR` *
`exL` i *	$H,\ \exists x{:}T.\ A,\ H' \vdash G$ $\quad H,\ x{:}T,\ A,\ H' \vdash G$	$H \vdash \exists x{:}T.\ A$ $\quad H \vdash A[t/x]$	`exR` t
`hypothesis` i	$H,\ A,\ H' \vdash A$	$H \vdash G$ $\quad H \vdash A$ $\quad H,\ A \vdash G$	`cut` A
`magic`	$H \vdash A\ \vee\ \neg A$	$H \vdash G$ $\quad H,\ A \vdash G$ $\quad H,\ \neg A \vdash G$	`ClassCases` A

* *Proviso*: If x already occurs in H or H', it must be renamed in the quantified formula before applying the rule.

2. **Equality**

$H,\ \vdash\ x = x$
by `reflexivity`

$H\ \vdash\ C[s/x]$
by `substitution` $s{=}t$ *
$\quad H\ \vdash\ s{=}t$
$\quad H\ \vdash\ C[t/x]$

*: if some but not all of several occurrences of s in the conclusion C shall be replaced, one has to view C as instance of a more general term $C[x]$ in which all the occurrences of x have been replaced by s (hence the notation $C[s/x]$) while other occurrences of s are "constant". In this case it becomes necessary to provide $C[x]$ as additional parameter to the rule.

3. **Library rules**

```
H ⊢ L
  by lemma "theorem-name"
```

```
H ⊢ C
  by cut_lemma "theorem-name"
    H, L ⊢ C
```

L is the statement of the library theorem "*theorem-name*"

4. **Arithmetic rules**:

```
H, i:int, H' ⊢ C
  by Induction i
    H, i:int, H', i<0, C[(i+1)/i] ⊢ C
    H, i:int, H' ⊢ C[0/z]
    H, i:int, H', 0<i, C[(i-1)/i] ⊢ C
```

```
H ⊢ C
  by arith *
```

*: `arith` is a decision procedure for elementary arithmetic. It proves the conclusion solely on the basis of investigating integer inequalities in H and C (see section 25.4).

```
H ⊢ C[f(i)/z]
  by ReduceDown
    H ⊢ C[t_d[i,f(i+1)]/z]
    H ⊢ i < 0
```

```
H ⊢ C[f(i)/z]
  by ReduceBase
    H ⊢ C[t_b/z]
    H ⊢ i = 0
```

```
H ⊢ C[f(i)/z]
  by ReduceUp
    H ⊢ C[t_u[i,f(i-1)]/z]
    H ⊢ 0 < i
```

The reduction rules apply only if f is defined inductively as

```
f(i) ≡    t_d[i,y]   if i<0, where y=f(i+1)
          t_b        if i=0
          t_u[i,y]   if i>0, where y=f(i-1)
```

5. **List rules**:

```
H, l:list, H' ⊢ C
  by ListInduction l
    H, l:list, H' ⊢ C[[]/l]
    H, l:list, H', C ⊢ C[x::l/l]
```

```
H ⊢ C[f([])/z]
  by Reduce
    H ⊢ C[t_b/z]
```

```
H ⊢ C[f(i::tl)/z]
  by Reduce
    H ⊢ C[t_u[i,tl,f(tl)]/z]
```

The reduction rules (which are less flexible than the ones for integers) apply only if f is defined inductively as

```
f(l) ≡    t_b           if l=[]
          t_u[i,tl,y]   if l=i::tl where y=f(tl)
```

25.4 The arithmetic decision procedure `arith`

`arith` can decide the validity of all sentences in the theory $\mathcal{A}$ of elementary arithmetic, which is a subtheory of λ-PRL

The theory $\mathcal{A}$ of elementary arithmetic

Formal Language: *Quantifier-free elementary arithmetic formulae*

Terms are built from integer constants, variables, and the function symbols `+`, `-`, `*`.

Atomic formulae are expressions of the form $t_1\,\rho\,t_2$, where t_1, t_2 are terms and ρ one of the predicate symbols $=, \neq, <, >, \leq, \geq$.

Formulas are built from atomic formulae and the boolean connectives $\neg$, $\wedge$, $\vee$ und $\Rightarrow$.

Axioms: *Limited equality axioms and the usual arithmetic axioms*

Equality. $\forall$x,y,z$\in\mathbb{Z}$

1. x=x (Reflexivity)
2. x=y $\Rightarrow$ y=x (Symmetry)
3. x=y $\wedge$ y=z $\Rightarrow$ x=z (Transitivity)
4. x=y $\wedge$ x ρ z $\Rightarrow$ y ρ z x=y $\wedge$ z ρ x $\Rightarrow$ z ρ y (Limited substitution)[2]

Axioms of constant arithmetic: 1+1=2, 2+1=3, 3+1=4, ...

Ring axioms. $\forall$x,y,z$\in\mathbb{Z}$

1. x+y = y+x x*y = y*x (Commutativity)
2. (x+y)+z = x+(y+z) (x*y)*z = x*(y*z) (Associativity)
3. x*(y+z) = (x*z)+(y*z) (Distributivity)
4. x+0 = x x*1 = x (Identities)
5. x+(-x) = 0 (Additive inverse)
6. x-y = x+(-y) (Definition of '-')

Axioms of discrete linear order. $\forall$x,y,z$\in\mathbb{Z}$

1. $\neg$(x<x) (Irreflexivity)
2. x<y $\vee$ x=y $\vee$ y<x (Trichotomy)
3. x<y $\wedge$ y<z $\Rightarrow$ x<z (Transitivity)
4. $\neg$(x<y $\wedge$ y<x+1) (Discreteness)

Definition of inequalities. $\forall$x,y,z$\in\mathbb{Z}$

1. x$\neq$y $\Leftrightarrow$ $\neg$(x=y)
2. x>y $\Leftrightarrow$ y<x
3. x$\leq$y $\Leftrightarrow$ x<y $\vee$ x=y
4. x$\geq$y $\Leftrightarrow$ y<x $\vee$ x=y

Monotonicity. $\forall$x,y,z,w$\in\mathbb{Z}$

1. x$\geq$y $\wedge$ z$\geq$w $\Rightarrow$ x+z$\geq$y+w (Addition)
2. x$\geq$y $\wedge$ z$\leq$w $\Rightarrow$ x-z$\geq$y-w (Subtraction)
3. x$\geq$0 $\wedge$ y$\geq$z $\Rightarrow$ x*y$\geq$x*z (Multiplication)
4. x>0 $\wedge$ x*y$\geq$x*z $\Rightarrow$ y$\geq$z (Cancellation/factoring)

[2] Only *closed* substitutions are allowed: x=z $\wedge$ x$\neq$x*y implies z$\neq$x*y but *not* z$\neq$z*y.

The `Arith` rule in refinement logic can be called as follows:

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

This 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.

The decision procedure essentially works as follows

1. *Transform* $H_1, \ldots H_m \vdash G_1 \vee \ldots \vee G_n$
   into $H_1, \ldots H_m, \ \neg G_1, \ldots, \neg G_n \vdash \texttt{false}$
   and consider from now on only the set of hypotheses.
2. *Split all conjuncts as with the rule* `andL`.
3. *Transform inequalities of the form* $x \neq y$ *into disjuncts* $x \geq y\texttt{+1} \vee y \geq x\texttt{+1}$ *and then split all disjuncts in the hypotheses as with the rule* `orL`.
   This generates several proof goals which all *must be investigated.*
4. *In atomic formulae replace all subterms that are not* $\mathcal{A}$*-terms by* new *variables (equal subterms will be replaced by the same variable).*
   Remove all hypotheses that are not atomic $\mathcal{A}$*-formulae.*
5. *Convert all the comparands of an atomic formula into standard polynomial expressions, (i.e. sums of products).*

   Convert the result into monadic linear polynomes of the form `c +` u_i *by replacing all non-constant parts by new variables.*
6. *Convert each atomic formula into the form* $t_1 \geq t_2$*, where* t_1 *is a variable* $\texttt{u}_i$ *or the constant* `0` *und* t_2 *a monadic linear polynome.*
7. *Create the ordering graph of the resulting formula set. Generate one node for each variable and constant and an edge* $u_i \xrightarrow{c} u_j$ *for the inequality* $\texttt{u}_i \geq \texttt{u}_j\texttt{+}c$*.*
8. *Check whether the graph has a positive cycle (standard algorithm)*

`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.

Example problems that can be solved with `Arith` are

- `0<x` $\vdash$ `0<x+2`
- H, `x`$\leq$`y`, `x`$\neq$`y` $\vdash$ `x<y`
- H, `x+y>z`, `2*x`$\geq$`z`, `x+y+2*x`$\geq$`z+z+1` $\vdash$ `3*x+y`$\geq$`2*z-1`

Goals like `0<x` $\vdash$ `0<x*x` or H, `x+y>z`, `2*x`$\geq$`z` $\vdash$ `3*x+y` $\geq$ `2*z-1`, however, require explicit applications of a `monotonicity` rule before calling `Arith`.

For details see Tat-hung Chan's article *An algorithm for checking PL/CV arithmetic inferences*" in R. Constable, S. Johnson, C. Eichenlaub, Introduction to the PL/CV2 Programming Logic, LNCS 135, Springer 182, p. 227–264.

Examples: Proofs, language extensions etc.

1. *Language extensions:*

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

   $$\begin{array}{lcll} l_1 \texttt{ @ } l_2 & \equiv & l_2 & \texttt{if } l_1\texttt{=[]}, \\ & & i\texttt{::}y & \texttt{if } l_1\texttt{=}i\texttt{::}l_1' \texttt{ where } y\texttt{=}l_1' \texttt{ @ } l_2. \end{array}$$

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

   $$\begin{array}{lcll} |l| & \equiv & \texttt{0} & \texttt{if } l\texttt{=[]} \\ & & y\texttt{+1} & \texttt{if } l\texttt{=}i\texttt{::}l' \texttt{ where } y\texttt{=}|l'| \end{array}$$

2. *A brief discussion of* `arith`*:*

   The PRL system provides a decision procedure for the subset of elementary arithmetic that consists of virtually all the trivial arithmetical insights that pop up so often during a proof. This rule, called `arith` in refinement logic is used to automatically solve problems like

   - `0<x` $\vdash$ `0<x+2`
   - H`,` `x`$\leq$`y, x`$\neq$`y` $\vdash$ `x<y`
   - H`,` `x+y>z, 2*x`$\geq$`z, x+y+2*x`$\geq$`z+z+1` $\vdash$ `3*x+y`$\geq$`2*z-1`

   The following briefly illustrates how the rule operates.

   (a) 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 `arith` fails.

   (b) The decision procedure essentially works as follows – Example:

      i. H`,` `x+y>z, 2*x`$\geq$`z, x+y+2*x`$\geq$`z+z+1,` $\neg$`(3*x+y`$\geq$`2*z-1)` $\vdash$ `false`

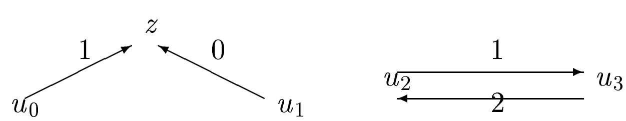
      ii. —

      iii. —

      iv. H disappears: `x+y>z, 2*x`$\geq$`z, x+y+2*x`$\geq$`z+z+1,` $\neg$`(3*x+y`$\geq$`2*z-1)` $\vdash$ `false`

      v. u_0`>z,` $u_1\geq$`z,` $u_2\geq$`1+`u_3`,` $\neg$`(`$u_2\geq$`(-1)+`u_3`)` $\vdash$ `false`
      with $u_0 \equiv$ `x+y,` $u_1 \equiv$ `2*x,` $u_2 \equiv$ `3*x+y` and $u_3 \equiv$ `2*z`

      vi. $u_0\geq$`z+1,` $u_1\geq$`z,` $u_2\geq$`1+`u_3`,` $u_3\geq u_2$`+2` $\vdash$ `false`
      Conversions are $\neg x\geq y \Leftrightarrow x<y$, $x>y \Leftrightarrow x\geq y$`+1`,
      x`=`$y \Leftrightarrow x\geq y \wedge y\geq x$, $x\neq y \Leftrightarrow x\geq y$`+1` $\vee$ $y\geq x$`+1`, *as well as adding constants*

      vii. 

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

   (c) `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` $\vdash$ `0<x*x` or H`,` `x+y>z, 2*x`$\geq$`z` $\vdash$ `3*x+y` $\geq$ `2*z-1`, however, require explicit applications of a `monotonicity` rule before calling `arith`.

3. *Proofs as programs:*

Using `arith` we are able to prove interesting theorems, for instance that every natural number has an integer square root.

```
⊢ ∀n:int.  0≤n  ⇒  ∃y:int.  y^2≤n  ∧  n<(y+1)^2                     by allR
|
n:int ⊢  0≤n  ⇒  ∃y:int.  y^2≤n  ∧  n<(y+1)^2                        by impR
\
 n:int, 0≤n ⊢ ∃y:int.  y^2≤n  ∧  n<(y+1)^2                           by Induction
 |\
 | n:int, 0≤n, n<0, ∃y:int. y^2≤n+1 ∧ n+1<(y+1)^2 ⊢ ∃y:int. y^2≤n ∧ n<(y+1)^2
 |                                                                   by arith
 |\
 | n:int, 0≤n ⊢ ∃y:int.  y^2≤0  ∧  0<(y+1)^2                         by exR 0
 | |
 | n:int, 0≤n ⊢ 0^2≤0  ∧  0<(0+1)^2                                  by arith
  \
   n:int, 0≤n, 0<n, ∃y:int. y^2≤n-1 ∧ n-1<(y+1)^2 ⊢ ∃y:int. y^2≤n ∧ n<(y+1)^2
   |                                                                 by exL 2 THEN andL 3
   |
   n:int, 0≤n, 0<n, y^2≤n-1, n-1<(y+1)^2 ⊢ ∃y:int. y^2≤n ∧ n<(y+1)^2
   |                                                                 by cut n<(y+1)^2 ∨ n=(y+1)^2
   |\
   | ...  ⊢ n<(y+1)^2 ∨ n=(y+1)^2                                    by arith
    \
     ..., n<(y+1)^2 ∨ n=(y+1)^2 ⊢ ∃y:int. y^2≤n ∧ n<(y+1)^2          by orL 5
     |\
     | ..., n<(y+1)^2 ⊢ ∃y:int. y^2≤n ∧ n<(y+1)^2                    by exR y
     | |
     | ..., n<(y+1)^2 ⊢  y^2≤n  ∧  n<(y+1)^2                         by andR
     | |\
     | | ..., n<(y+1)^2 ⊢  y^2≤n                                     by arith
     |  \
     |   ..., n<(y+1)^2 ⊢  n<(y+1)^2                                 by hypothesis 6
      \
       ..., n=(y+1)^2 ⊢ ∃y:int. y^2≤n ∧ n<(y+1)^2                    by exR y+1
       |
       ..., n=(y+1)^2 ⊢ (y+1)^2≤n ∧ n<(y+1+1)^2                      by andR
       |\
       | ..., n=(y+1)^2 ⊢ (y+1)^2≤n                                  by arith
        \
         ..., n=(y+1)^2 ⊢ n<(y+1+1)^2                                by arith
```

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.