25 λ-PRL: Towards Program Verification

In the previous lectures we have explored the theoretical limitations of mathematical reasoning within formal logical theories. We will now turn back to the applied aspects of formal logics and introduce a formalism that is well suited for reasoning about programming and a large fragment of arithmetic.

The formalisms used so far were based on axiomatizations of various fragments of mathematics, such as equality, orderings, functions, algebraic structures, induction, and the Peano axioms. That allowed us to keep first-order logic as the basic formalism, but had the disadvantage that our reasoning apparatus is very primitive, which means that formal proofs of significant results will become incredibly tedious.

The formal system that we want to present now provides a more practical approach. It considers certain fragments of mathematics, such as integers and lists of integers, as primitives of the logic and includes rules for reasoning about these primitives directly. Furthermore, it includes a extra-logical features such as a *definition mechanism*, *decision procedures*, and *proof tactics*. These features simplify the tasks of formalizing mathematical concepts and formally proving theorems and make formal reasoning about certain fragments of mathematics feasible.

The formalism resulting from these considerations is called **P**roof **R**efinement **L**ogic. Because of the extra-logical features, it is not a pure proof calculus but a hybrid system that includes proof search algorithms in some of its rules. In this course we will discuss λ-PRL, a logic for reasoning about programs, integers and lists of integers. In contrast to the logic so far, λ-PRL's logic is *typed*, that is variables are associated with a type when they are bound by a quantifier. The type system is fairly simple and includes only integers and lists of integers. λ-PRL's successor Nuprl is based on a much more elaborate *type theory*, which is capable of expressing all data structures used in mathematics and programming. The proof calculus of PRL is a sequent calculus similar to the refinement logic defined earlier.

In the following we briefly summarize the formal language of λ-PRL, mechanisms for defining new language expressions in terms of existing ones, the proof calculus of λ-PRL, and some of the decision procedures that are built into the logic.

25.1 The Formal Language of λ-PRL

The formal language of λ-PRL distinguishes types, (object) variables, integer and list constants, as well as predicate and function symbols. The latter will always be *associated with types*. For instance, the function `+` is associated with the type `int`$\times$`int` $\rightarrow$ `int` and will not accept constants or variables of a different type as input.

Terms are built from constants, variables, function applications and application of infix operators. *Atomic formulas* are built by applying either an equality or an order relation to two terms. *Formulas* are built from atomic formulas in the usual way, except for the fact that quantifiers are typed. Below is a detailed account of the formal language.

- **Types**: `int` and `list`
- **Variables**: $a, b, c, x, y, ...$
- **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` → `int`
  (Negation)

  – The binary infix operators `+`, `-`, `*`, `/`, `mod` `int`×`int` → `int`
  (addition, subtraction, multiplication, division, quotient remainder)

  – The binary infix operator `::` `int`×`list` → `list`
  (prepending an element to a list)

  – The unary function symbol `hd` and `tl` `list` → `int` and `list` → `list`
  (head and tail of a list)

  Constant predicate symbols: *associated with types*
  – The atomic predicate symbols `true` and `false`
  – The binary infix predicate symbols `=` and `<` on `list`×`list` and `int`×`int`
  (equality on integers and lists, strict order on integers)[5]

- **Terms** are:
  – Constants and variables
  – Function applications $f(t_1,\ldots,\ t_n)$ — 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 $[t_1,\ldots,\ t_n]$ — terms t_i must have type `int`

- **Formulas** are:

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

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

Note that the notation for quantifiers deviates somewhat from the one used so far.

[5]The predicate symbols `>`, $\leq$, $\geq$ can be defined through language extension.

Language Extensions

The formal language of λ-PRL may be extended by user-defined concepts through *definitions*. These are templates for introducing a new notation for an already existing expression through *textual replacements*. We write definitions in the form

template with formal parameters $\equiv$ *expression*

For instance, we can introduce a less-equal predicate and the unique-existence quantifier through the following definitions

$$
\begin{array}{lcl}
x{\le}y & \equiv & x\texttt{=}y \;\lor\; x\texttt{<}y \\
x^2 & \equiv & x{*}x \\
\exists!x{:}T.\,P[x] & \equiv & \exists x{:}T.\,P[x] \;\land\; \forall \mathtt{y}{:}T.\,P[\mathtt{y}/x] \supset x\texttt{=}\mathtt{y}
\end{array}
$$

In a definition, the expressions in math-font (x, T, $P[x]$) are formal parameters of the template and the expression, while `y` is not.[6] 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` $\lor$ `4<5` and $\exists$`!x:int. (x<5` $\land$ `3<x)` to $\exists$`x:int. (x<5` $\land$ `3<x)` $\land$ $\forall$`y:int. (y<5` $\land$ `3<y)` $\supset$ `x=y`.
The expression on the right hand side of a definition can be any term and formula. In addition to that λ-PRL supports *recursively defined functions*, which are declared by a *function definition block*:[7]

$$
f(x_1{:}T_1,\ldots,x_n{:}T_n){:}T \;\equiv\; \begin{array}{l}
\mathit{case}_1 \;\texttt{=>}\; t_1 \\
\mathit{case}_2 \;\texttt{=>}\; t_2 \\
\quad\vdots \\
\mathit{case}_m \;\texttt{=>}\; t_m
\end{array}
$$

In a block, the cases may either be variables or integer constants. The variable must correspond to the first argument of the function. It expresses the step case of the inductive definition and is used in the first and last case (down- and upgoing induction) when recursing over integers and in the last case only when recursing over lists. The remaining cases describe the function's value for the base cases, which are represented by constants in consecutive order. The definition of the factorial, for instance, could be expressed as follows.

```
factorial(n:int):int ≡ n => 1
                       0 => 1
                       n => factorial(n-1)*n
```

[6] In the implemented systems, parameters are marked by angle brackets, so the above definition would read as $\langle\mathrm{x}\rangle{\le}\langle\mathrm{y}\rangle \;\equiv\; \langle\mathrm{x}\rangle{=}\langle\mathrm{y}\rangle \;\lor\; \langle\mathrm{x}\rangle{<}\langle\mathrm{y}\rangle$

[7] The definition of recursive functions in λ-PRL is different from the one in its successor Nuprl. In Nuprl, a recursive definition is a part of the formal language, that is there is a *term* for inductively defined concepts on integers and lists. In λ-PRL recursion requires a specialized mechanisms that does not belong to the logic itself. This makes it easier to define recursive functions but requires a special mechanism for reasoning about inductively defined concepts that does not generalize well.

In cases where a function does not recurse over its first argument, we may use the definition mechanism to rearrange the order of arguments afterwards. For instance, exponentiation x^i can be defined as follows.

```
exp(i:int,x:int):int ≡ i => 0
                       0 => 1
                       i => exp(i-1,x)*x

x^i                  ≡ exp(i,x)
```

If the function recurses over lists, the integer constants represent the length of the list. So the length of a list l and selecting the i-th element of l can be defined as follows.

```
length(l:list):int       ≡ 0 => 0
                           l => length(tl(l))+1

index(i:int,l:list):int  ≡ 0 => hd(l)
                           i => index(i-1,tl(l))
```

Note that `index` returns a default value if i is out of range.

Functions may also be defined by *mutual recursion*. In this case two or more function definition block are declared at the same time and each function definition has exactly the same number of cases.

Functions may also be defined via *extraction*, since any proof of a theorem of the form $\forall x_1..x_n{:}T.\ \exists y{:}T'.A$, implicitly contains an algorithm for a function $f{:}T^n{\rightarrow}T'$ with the property $\forall t_1..t_n{:}T.\ A[t_1..t_n, f(t_1..t_n)/x_1..x_n, y]$.

The proof of the formula $\forall$`n:int.(0`$\leq$`n` $\supset$ $\exists$`y:int.(y`$^2\leq$`n` $\wedge$ `n<(y+1)`^{2}`))` in Figure 1, for instance, proves for every nonnegative integer there is an integer square root. In the proof we implicitly describe a method for constructing the integer square root recursively. In the negative step case there is no solution (the condition `0`$\leq$`n` excludes this case). In the base case `n=0` we provide the solution `y=0`. In the step case we analyze the solution for `n-1` and either reuse it or increase it by one. In λ-PRL, this method would be described as follows

```
sqrt(n:int):int ≡ n => 0
                  0 => 0
                  n => if (sqrt(n-1)+1)^2<n then sqrt(n-1) else sqrt(n-1)
```

where the conditional `if` $i{\leq}j$ `then` a `else` b is formalized as

```
if_lt0(i:int,a:T,b:T)  ≡ i => a,  0 => b,  i => b

if i<j then a else b   ≡ if_lt0(i-j,a,b)
```

The PRL systems provide a mechanism for extracting an algorithm from the formal proof and assigning it to a function name. In λ-PRL, executing a check-command will make this assignment. Later versions (like Nuprl) provide a special term for this purpose, which allows using the following definition to link the extract term to a function name.

f ≡ `extract_of "` *theorem-name*`"`

where *theorem-name* is the name of the theorem from which the function will be extracted. We will use the above form in the rest of this course.

```
⊢ ∀n:int.  0≤n ⊃ ∃y:int.  y^2≤n  ∧  n<(y+1)^2                BY induction 1 0 0 1
|\
| n:int, n<0, (0≤n+1 ⊃ ∃y:int.  y^2≤n+1  ∧  n+1<(y+1)^2)
| ⊢ 0≤n ⊃ ∃y:int.  y^2≤n  ∧  n<(y+1)^2                        BY impR
| |
| n:int, n<0, (0≤n+1 ⊃ ∃y:int.  y^2≤n+1  ∧  n+1<(y+1)^2), 0≤n
| ⊢ ∃y:int.  y^2≤n  ∧  n<(y+1)^2                              BY arith  √
|\
| ⊢ 0≤0 ⊃ ∃y:int.  y^2≤0  ∧  0<(y+1)^2                        BY impR
| |
| 0≤0 ⊢ ∃y:int.  y^2≤0  ∧  0<(y+1)^2                          BY exR 0
| |
| 0≤0 ⊢ 0^2≤0  ∧  0<(0+1)^2                                   BY arith  √
 \
  n:int, 0<n, (0≤n-1 ⊃ ∃y:int. y^2≤n-1 ∧ n-1<(y+1)^2)
  ⊢ 0≤n ⊃ ∃y:int. y^2≤n ∧ n<(y+1)^2                           BY impR
  |
  n:int, 0<n, (0≤n-1 ⊃ ∃y:int. y^2≤n-1 ∧ n-1<(y+1)^2), 0≤n
  ⊢ ∃y:int. y^2≤n ∧ n<(y+1)^2                                 BY impL 3
  |\
  | n:int, 0<n, ..., 0≤n  ⊢ 0≤n-1                             BY arith  √
   \
    n:int, 0<n, 0≤n-1, ∃y:int. y^2≤n-1 ∧ n-1<(y+1)^2, 0≤n
    ⊢ ∃y:int. y^2≤n ∧ n<(y+1)^2                               BY exL 4
    |
    n:int, 0<n, 0≤n-1, y:int, y^2≤n-1 ∧ n-1<(y+1)^2, 0≤n
    ⊢ ∃y:int. y^2≤n ∧ n<(y+1)^2                               BY andL 5
    |
    n:int, 0<n, 0≤n-1, y:int, y^2≤n-1, n-1<(y+1)^2, 0≤n
    ⊢ ∃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 8
      |\
      | ..., 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 8  √
       \
        ..., 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  √
```

Figure 1: Proof of the Integer Square Root Problem

25.2 The Proof Calculus

The proof calculus of λ-PRL is an extension of the refinement logic introduced in Lecture 11. It provides rules for predicate logic, equality, arithmetical reasoning, reasoning about lists, and structural rules such as a cut rule and a rule for consulting previously proven lemmata.

Since the formal language of λ-PRL includes types, it is necessary to include *type declarations* in the hypotheses of a sequent. A declaration of the form $x{:}T$ denotes that x is declared to be of type T.

25.2.1 Propositional Logic

The logical rules are slightly more flexible than the rules of refinement logic, because the deal with iterations in a single step.

Conjunction

$H,\ A_1 \wedge \ldots \wedge A_n,\ H' \vdash G$
by andL i

$H,\ A_1,\ \ldots,\ A_n,\ H' \vdash G$

$H \vdash A_1 \wedge \ldots \wedge A_n$
by andR

$H \vdash A_1$
$\vdots$
$H \vdash A_n$

Disjunction

$H,\ A_1 \vee \ldots \vee A_n,\ H' \vdash G$
by orL i

$H,\ A_1,\ H' \vdash G$
$\vdots$
$H,\ A_n,\ H' \vdash G$

$H \vdash A_1 \vee \ldots \vee A_n$
by orR i

$H \vdash A_i$

Implication

$H,\ A \supset B,\ H' \vdash G$
by impL i

$H,\ A \supset B,\ H' \vdash A$
$H,\ H',\ A,\ B \vdash G$

$H \vdash A \supset B$
by impR

$H,\ A \vdash B$

Negation

$H,\ \sim A,\ H' \vdash G$
by notL i

$H,\ \sim A,\ H' \vdash A$

$H \vdash \sim A$
by notR

$H,\ A \vdash$ false

False and True

$H,$ false$,\ H' \vdash G$
by falseL i

$H,\ A,\ \sim A,\ H' \vdash$ false
by falseR

$H \vdash$ true
by trueR

25.2.2 Quantifier Rules

H, $\forall x_1..x_n{:}T.\,A$, $H' \vdash G$
by allL i $t_1..t_n$

H, $\forall x_1..x_n{:}T.\,A$, H', $A[t_1.t_n/x_1.x_n] \vdash G$

$H \vdash \forall x_1..x_n{:}T.\,A$
by allR

H, $x_1{:}T, \ldots x_n{:}T \vdash A$

H, $\exists x_1..x_n{:}T.\,A$, $H' \vdash G$
by exL i

H, $x_1{:}T, \ldots x_n{:}T$, A, $H' \vdash G$

$H \vdash \exists x_1..x_n{:}T.\,A$
by exR $t_1..t_n$

$H \vdash A[t_1.t_n/x_1.x_n]$

Each t_i must be a well-formed term of type T. The newly declared x_i will be renamed if they do already occur in the hypotheses list.

25.2.3 Induction

$H \vdash \forall x{:}\texttt{int}.\,A$
by induction d lo hi u

H, $x{:}\texttt{int}$, $x{<}lo$, $A[(x{+}d)/x] \vdash A$
$H \vdash A[lo/x]$
$\vdots$
$H \vdash A[hi/x]$
H, $x{:}\texttt{int}$, $hi{<}x$, $A[(x{-}u)/x] \vdash A$

$H \vdash \forall x{:}\texttt{list}.\,A$
by induction x_1::...::x_n::l

$H \vdash A[\texttt{[]}/x]$
H, $y_1{:}\texttt{int} \vdash A[\texttt{[}x_1\texttt{]}/x]$
$\vdots$
H, $x_1{:}\texttt{int},..,x_n{:}\texttt{int} \vdash A[\texttt{[}x_1,\ldots,x_n\texttt{]}/x]$
H, $x_1{:}\texttt{int},..,x_n{:}\texttt{int}, l{:}\texttt{list}$, $A[l/x]$
$\vdash A[x_1{::}\ldots{::}x_n{::}l/x]$

25.2.4 Structural Rules

H, A, $H' \vdash$ A
by hyp i

$H \vdash G$
by cut $A_1..A_n$

$H \vdash A_1$
$\vdots$
$H, A_1..A_{n-1} \vdash A_n$
$H, A_1..A_n \vdash G$

$H \vdash L$
by lemma "*theorem-name*"

$H \vdash C$
by cut_lemma "*theorem-name*"

H, $L \vdash C$

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

25.2.5 Decision Procedures

In contrast to the above rules, decision procedures are well-understood algorithms for completely solving problems within a decidable domain. They are of tremendous value for practical theorem proving, as they solve goals in a single step where a proof entirely based on primitive rules would require hundreds of steps to complete. λ-PRL includes the following decision procedures.

$H \vdash C$
by `division`
$H \vdash C'$

$H \vdash C$
by `arith`

$H \vdash C$
by `equality`

$H \vdash C$
by `simplify` $t_1..t_n$ `in` $i_1..i_k$
$H' \vdash C'$

`division` summarizes some of the basic laws about the interplay between division and quotient remainder and the relations `=` and `<`. The remaining subgoal describes the preconditions for C. For instance, the precondition for `0<a/b` is `(0<b≤a) ∨ (a≤b<0)`.

`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 26.1.1).

`equality` proves the conclusion solely on the basis of reflexivity, symmetry, transitivity, substitution in functions, and the equalities `hd(`i`::`l`)=`i, `tl(`i`::`l`)=`l, and `~(`i`::`l`=[])`.

`simplify` simplifies the terms $t_1..t_n$ in the hypotheses $i_1..i_k$ (where hypothesis 0 is the conclusion) by evaluating applications of constant functions and operators, of defined functions, and functions extracted from proofs. The term `hd((i*(3*0+1)))::`l, for instance, is simplified to i and `tl((i*(3*0+1)))::`l is simplified to l.

25.2.6 Function Definition

$H \vdash C$
by `def` $f(t_1..t_n)$
$H,\ f(t_1..t_n) = t \vdash C$

$H \vdash C$
by `def` $f(t_1..t_n)$ `up`
$H, \vdash upper{<}t_1$
$H,\ upper{<}t_1,\ f(t_1..t_n) = t \vdash C$

If t_1 is a constant within the base range of the definition of f then t is the corresponding result, where the other arguments of f are instantiated by the t_i.

Otherwise a direction for the induction must be given as additional argument to the rule. If the direction is `up`, then *upper* is the highest base case in the definition of f and the function is instantiated accordingly. The case for `down` is similar.

If f is extracted from the proof of a theorem of the form $\forall x_1..x_n{:}T.\ \exists y{:}T'.A$ then applying the rule produces the goal.

$H \vdash C$
by `def` $f(t_1..t_n)$
$H,\ A[t_1..t_n, f(t_1..t_n)/x_1..x_n, y] \vdash C$

25.4 Program Extraction

λ-PRL provides a mechanism for extracting algorithms from formal proofs. This mechanism uses the fact that most inference rules of λ-PRL's proof calculus can be associated with the construction of *evidence* and that the proof tree can be considered as instruction for constructing evidence for the main goal from evidence for the proof leaves.

The rule andR for dealing with a conjunction in the conclusion of a sequent, for instance, decomposes a sequence $H \vdash A_1 \wedge \ldots \wedge A_n$ into the sequences $H \vdash A_1 \ldots H \vdash A_n$. If $a_1 \ldots a_n$ are the evidences for the validity of $A_1, \ldots A_n$, then the tuple $\langle a_1, \ldots, a_n \rangle$ can be considered as evidence for the $A_1 \wedge \ldots \wedge A_n$. In the same way we can associate all other rules of λ-PRL with specific methods for building evidence.

orR: If a_i is the evidence for A_i, then $\langle i, a_i \rangle$ is sufficient evidence for $A_1 \vee \ldots \vee A_n$, since it indicates that the i-th disjunct is true with evidence a_i.

impR: If b is evidence for B whenever a is evidence for A, then $\lambda a.b$, the function that maps a into b, is evidence for $A \supset B$

allR: If $a[x_1, .., x_n]$ is evidence for A whenever the x_i are of type T then $\lambda x_1, .., x_n . a[x_1, .., x_n]$ is evidence for $\forall x_1 . . x_n : T . A$

exR: If $t_1, \ldots, t_n$ are elements of type T and the predicate A holds for $t_1, \ldots, t_n$ then the tuple $\langle t_1, \ldots, t_n, a[t_1, .., t_n / x_1, .., x_n] \rangle$ is evidence for $\exists x_1 . . x_n : T . A$

andL: If g is evidence for G whenever $a_1 \ldots a_n$ are evidences for $A_1, \ldots A_n$ then $g[\langle a_1, \ldots, a_n \rangle / a]$ is evidence for G whenever a is evidence for $A_1 \wedge \ldots \wedge A_n$.

$\vdots$

Similar explanations can be given for all other inference rules. The induction rules construct recursive functions, the hyp rule links the required evidence for the conclusion to existing evidence in the hypothesis. The cut rule allows assembling evidence for the stated goals A_i and then using this evidence in the main proof. The decision procedures provide no computationally meaningful evidence, as they only assert the truth of a goal sequent.

λ-PRL's extraction mechanism analyzes a proof tree and builds evidence for all its nodes bottom up, i.e. starting at the leaves and going up to the root, which represents the proof of the main goal. In contrast to later PRL systems, where the extraction mechanism is closely tied to a very elaborate proof calculus, *λ-PRL only allows extraction from theorems of the form* $\forall x_1 . . x_n : T . \exists y : T' . A$. It constructs the evidence from the proof, and converts it into a function $f : T^n \rightarrow T'$ that satisfies the property $\forall t_1 . . t_n : T . A[t_1 . . t_n, f(t_1 . . t_n) / x_1 . . x_n, y]$. Note that the evidence for $A[t_1..t_n, f(t_1..t_n)/x_1..x_n, y]$ is dropped during the extraction process, as it is not relevant for the evaluation of the function.

The ability to extract programs from proofs is one of the strongest reasons for using a single-conclusioned refinement logic instead of a multi-conclusioned sequent calculus. While the latter often makes it easier to prove the validity of formulas, it is not possible to associate rules that generate more than one conclusion with the construction of (deterministic) evidence. The evidence for $\vdash A, B$ would only indicate that one of A or B must be valid, without identifying the formula that is actually true. Such evidence would be useless for the extraction of algorithms from proofs.