26 Proof Automation in λ-PRL

The inference system of λ-PRL enables us to prove significant amounts of non-trivial mathematics and especially theorems about algorithms and their properties. To a certain degree, this is due to the expressiveness of the logical calculus. The real strength of λ-PRL, however, comes from the system's ability to automate the development of proofs. Proof automation enables users to solve intellectually trivial, but formally tedious proofs in only a few steps and helps them structure the proofs in a way that makes the central proof ideas visible.

λ-PRL offers two forms of proof automation. *Decision procedures* use well-understood algorithms to solve problems within a decidable domain and are implemented as primitive inference rules of the system. Proof *tactics*, on the other hand, are meta-level programs that control the application of inference rules in a user-defined manner and thus add a lot of flexibility to the proof system. In the following we will discuss both features.

26.1 Decision procedures

In mathematical reasoning one quite often encounters problems that are well explored but have lengthy and complicated proofs in a formal system. Often, it is only interesting to find out whether they are true or not, while the individual proof steps do not lead to any interesting insights. For dealing with such problems, decision procedures tailored to reasoning about a particular domain offer substantial support for a formal system. They usually transform the problem into a problem over another domain, such as the problem of finding loops in a graph, for which there are well-known efficient algorithmic solutions. From the perspective of the formal proof system, they appear as a single, yet sophisticated inference rule. Obviously, the decision procedure has to be consistent with the rest of the formal system, i.e. the provability in domain it decides must match the corresponding notion of provability in the (larger) logical calculus. In this section we will describe two of the decision procedures that are used within λ-PRL, the `arith` rule for deciding elementary arithmetic (i.e. the `arith` rule), and the `equality` rule for basic equality reasoning.

26.1.1 Deciding Elementary Arithmetic

Arithmetical reasoning is a typical example of formal reasoning that needs to be supported by a decision procedure. There are several reasons for that. First, arithmetical reasoning occurs in almost all proofs and is utterly trivial from an intellectual point of view. Second, arithmetical insights come in an astonishing variety. There are hundreds of lemmata expressing essentially the same idea. For instance, the sequents `x+1<y, 0<t` $\vdash$ `(x+1)*t<y*t` and `x<y, 0<t` $\vdash$ `x*t<y*t` are syntactically different sequents although the first is a simple variation of the second. Finally, formal proofs for even the simplest arithmetical facts like "*if three integers differ at most one from each other then two of them are equal*" are fairly complicated if one has to rely on primitive inference rules.

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 $\sim, \wedge, \vee, \supset$.

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

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

1. x=x		(Reflexivity)
2. x=y $\supset$ y=x		(Symmetry)
3. x=y $\wedge$ y=z $\supset$ x=z		(Transitivity)
4. x=y $\wedge$ x ρ z $\supset$ y ρ z	x=y $\wedge$ z ρ x $\supset$ z ρ y	(Limited substitution)[8]

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

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

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

Figure 2: The theory $\mathcal{A}$ of elementary arithmetic

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

Addition				
	z>w	z≥w	z=w	z≠w
x>y	x+z≥y+w+2	x+z≥y+w+1 x+w≥y+z+1	x+z≥y+w+1	-----
x≥y	x+z≥y+w+1	x+z≥y+w x+w≥y+z	x+z≥y+w	-----
x=y	x+z≥y+w+1 y+z≥x+w+1	x+z≥y+w y+z≥x+w	x+z=y+w x+w=y+z	x+z≠y+w x+w≠y+z
x≠y	-----	-----	x+z≠y+w x+w≠y+z	-----

Multiplication				
	y≥z	y>z	y=z	y≠z
x>0	x*y≥x*z	x*y>x*z	x*y=x*z	x*y≠x*z
x≥0	x*y≥x*z	x*y≥x*z	x*y=x*z	-----
x=0	x*y=x*z x*y=0	x*y=x*z x*y=0	x*y=x*z x*y=0	x*y=x*z x*y=0
x≤0	x*y≤x*z	x*y≤x*z	x*y=x*z	-----
x<0	x*y≤x*z	x*y<x*z	x*y=x*z	x*y≠x*z
x≠0	-----	x*y≠x*z	x*y=x*z	x*y≠x*z

Subtraction				
	z>w	z≥w	z=w	z≠w
x>y	x-w≥y-z+2	x-w≥y-z+1 x-z≥y-w+1	x-w≥y-z+1	-----
x≥y	x-w≥y-z+1	x-w≥y-z x-z≥y-w	x-w≥y-z	-----
x=y	x-w≥y-z+1 y-w≥x-z+1	x-w≥y-z y-w≥x-z	x-w=y-z y-w=x-z	x-w≠y-z x-z≠y-w
x≠y	-----	-----	x-w≠y-z x-z≠y-w	-----

Cancellation				
	x*y>x*z	x*y≥x*z	x*y=x*z	x*y≠x*z
x>0	y>z	y≥z	y=z	y≠z
x<0	y<z	y≤z	y=z	y≠z
x≠0	y≠z	-----	y=z	y≠z

Figure 3: Variants of the monotonicity axioms

Thus proving significant mathematical results with a formal proof system becomes infeasible, unless there is substantial support for arithmetical reasoning. On the other hand, we know that arithmetic in general is undecidable. Therefore, decision procedures can only be developed for a *restricted arithmetic* that covers most of the common problems.

The theory $\mathcal{A}$ that underlies λ-PRL's `arith` procedure, describes what has been called *elementary arithmetic*. It is an *induction-free* theory upon constant arithmetic with addition, subtraction, multiplication, equality, and the usual integer inequalities. Its formal language and axioms is described in Figure 2. Logically, the theory $\mathcal{A}$ it is *quantifier-free* – all variables are assumed to be universally quantified.

Because of these restrictions, the theory is known to be decidable. However, a practically useful decision procedures needs to be reasonably efficient. For that reason, the laws of evaluating and simplifying arithmetical operations with integer constants and many variants of the theory's axioms, have to be directly incorporated into a procedure. Figure 3 describes the variants of the monotonicity axioms of $\mathcal{A}$ modulo the ring axioms. The rows and columns of the tables are indexed by term schemas. Every entry contains the conclusions from the hypotheses corresponding to its row and column.

Since every formula in restricted arithmetic has an equivalent one in conjunctive normal form and all conjunctions can be dealt with separately, it is sufficient to have `arith` deal only with goals of the form $H \vdash G_1 \vee \ldots \vee G_n$ where each of the G_i are *atomic formulas* in $\mathcal{A}$. As usual, the case $n = 0$ means that the conclusion is `false`.

We will explain the decision procedure by a running example. Consider the goal

H, i: `x+y>z`, H', j: `2*x≥z`, $H'' \vdash$ `3*x+y≥2*z-1`

where i: *hyp* denotes that i is the index of the hypothesis *hyp*.

1. arith begins by dealing with monotonicities, i.e. by adding conclusions that can be derived from the monotonicity axioms to the list of hypotheses hypotheses. While some of these steps can be performed purely automatically, others need some guidance by the user
   A *trivial monotonicity* is an application of a monotonicity axiom for addition where one of the hypotheses has the form $n\rho m$ where m and n are integer constants and ρ is one of $=$, $\neq$, $<$, $>$, $\leq$, $\geq$. All other monotonicities are *nontrivial*. Intuitively, a trivial monotonicity corresponds to adding constants on both sides of a relation in a meaningful way. For instance, we may conclude x+2≠y+4 from x=y but we cannot draw any conclusions about x+2 and y+4 from x≠y.
   In our example the monotonicity is nontrivial. We want to *add* hypotheses i and j and provide this information as argument to the arith rule, i.e.by calling arith $i+j$. As result arith adds a new hypothesis to the sequent and we get.

   H, i: x+y>z, H', j: 2*x≥z, H'', x+y+2*x≥z+z+1 ⊢ 3*x+y≥2*z-1

2. Since $\mathcal{A}$ is known to be decidable, we may use an indirect proof even when arith is used in a proof system that is considers the computational content of proofs (like λ-PRL or Nuprl). That is, instead of proving the sequent $H \vdash G_1 \vee \ldots \vee G_n$, it is sufficient to prove the sequent H, $\sim G_1, \ldots, \sim G_n \vdash$ false. As result of this transformation we get

   H, i: x+y>z, H', j: 2*x≥z, H'', x+y+2*x≥z+z+1, ∼(3*x+y≥2*z-1) ⊢ false

3. Next, arith splits all conjunctions in the hypotheses into atomic formulas by applying the same mechanism as in the rule andL. Our example sequent does not contain any conjunctions and thus remains unchanged.

4. In the hypotheses we may safely ignore all those which are not arithmetic formulas since these will not contribute to a arithmetic solution of the problem. In every hypothesis we replace all subterms that are not $\mathcal{A}$-terms by a variable and then remove all hypotheses that are not atomic $\mathcal{A}$-formulas. This gives us

   x+y>z, 2*x≥z, x+y+2*x≥z+z+1, ∼(3*x+y≥2*z-1) ⊢ false

5. Since inequalities of the form $x\neq y$ cannot be handled by the remaining procedure, they are converted into the disjunction $x\geq y$+1 ∨ $y\geq x$+1 and then split into atomic formulas, applying the same mechanism as in the rule orL. Thus arith generates two sequents per unequality, which means that arith *is exponential in the number of ≠-symbols*. In most cases, however, there are at most one or two ≠-symbols and therefore the *average complexity* of arith is low. In our example, there is no unequality and the sequent does not change.

6. In the next step arith normalizes the comparands in each formula into the standard representation of polynomials by sums of products $c + a_1{*}x_1{*}x_2 \ldots + \ldots + a{*}y_1^n{*}y_2^n \ldots$. This is a standard arithmetical conversion that causes equal terms to become syntactically identical. In our example we get

   x+y>z, 2*x≥z, 3*x+y≥1+2*z, ∼(3*x+y≥(-1)+2*z) ⊢ false

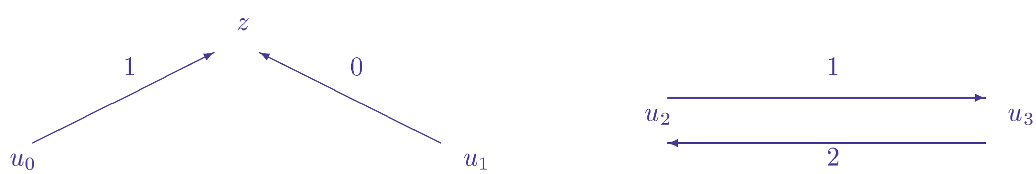
7. Now all comparands are transformed into *monadic linear polynomials*, i.e. expressions of the form $c+u_i$ where u_i is a variable and c a constant. This means that all nonconstant components are treated as variables. It can be shown that the resulting sequent is contradictory if and only if the original one has been. In the example we select $\mathtt{u}_0 \equiv$ `x+y`, $\mathtt{u}_1 \equiv$ `2*x`, $\mathtt{u}_2 \equiv$ `3*x+y`, and $\mathtt{u}_3 \equiv$ `2*z`, which results in

   $\mathtt{u_0>z,\ u_1\geq z,\ u_2\geq 1+u_3}\ \sim\mathtt{(u_2\geq(-1)+u_3)} \vdash \mathtt{false}$

8. Now all hypotheses are converted into the inequalities of the form $t_1 \geq t_2$ where t_1 is either a variable or the constant `0` and t_2 is a monadic linear polynomial. This can be done via standard conversions like x=$y \mapsto x\geq y \wedge y\geq x$, $\sim(x\geq y) \mapsto x<y$, and $x>y \mapsto x\geq y$+1. In our case the result is

   $\mathtt{u_0\geq z+1,\ u_1\geq z,\ u_2\geq 1+u_3,\ u_3\geq u_2+2} \vdash \mathtt{false}$

9. In the final step we analyze the set of comparisons in the hypotheses and search for a contradiction by trying to prove that some u_i must be greater than itself. For this purpose we translate the hypothesis list into a graph representing the order on the variables. A node represents variables or constants and an edge $u_i \xrightarrow{2} u_j$ indicates $\mathtt{u_i\geq u_j+2}$. The resulting graph has a positive cycle if and only if the list of hypotheses is contradictory, which is the case if and only if the original sequent is valid. Standard algorithms can detect positive cycles in a graph in quadratic time. In our example the resulting graph is

   and there is a positive cycle involving the nodes $\mathtt{u}_2$ and $\mathtt{u}_3$. Therefore the original sequent is true and `arith` succeeds.

Figure 4 summarizes the algorithm used by the `arith` rule. Here are a few examples of problems that can be handled by `arith` and would have been quite difficult to prove without it.

$\vdash$ `x-5 < x+10`	`BY arith`
i: $\mathtt{x\leq y}$, j: $\mathtt{x\neq y} \vdash$ `x<y`	`BY arith`
i: `0<x` $\vdash$ `0<x+2`	`BY arith`
i: `0<x` $\vdash$ `0<x*x`	`BY arith` i*i
i: $\mathtt{x+y\leq z}$, j: $\mathtt{y\geq 1} \vdash$ `x<z`	`BY arith` i-j
i: `x<x*x`, j: $\mathtt{x\neq 0} \vdash \mathtt{x\geq 2}$ | `x<0`	`BY arith` i/j
i: `x<y`, j: `0<z` $\vdash$ `x*z<y*z`	`BY arith` i*i
i: `x+y>z`, j: $\mathtt{2*x\geq z} \vdash \mathtt{3*x+y \geq 2*z-1}$	`BY arith` i+i

For further 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.

Given a goal of the form $H_1,\ldots,H_m \vdash G_1 \vee \ldots \vee G_n$ the algorithm for `arith` i *op* j, where *op* is one of `+, -, *, /` performs the following steps:

1. *Perform monotonicity steps as requested and create new hypotheses according to the table for op in Figure 3,*
2. *Transform* $H_1,\ldots H_m \vdash G_1 \vee \ldots \vee G_n$ *into* $H_1,\ldots H_m, \sim G_1,\ldots,\sim G_n \vdash$ `false` *and consider from now on only the set of hypotheses.*
3. *Split all conjuncts as with the rule* `andL`.
4. *In atomic formulas replace all subterms that are not* $\mathcal{A}$*-terms by* new *variables and then remove all hypotheses that are not atomic* $\mathcal{A}$*-formulas.*
5. *Transform inequalities of the form* $x \neq y$ *into* $x \geq y+1 \vee y \geq x+1$ *and then split all disjuncts in the hypotheses as with the rule* `orL`. *Investigate* all *resulting proof goals separately.*
6. *Convert the comparands of an atomic formula into standard polynomial expressions.*
7. *Convert the result into monadic linear polynomials, replacing non-constant parts by new variables.*
8. *Convert each atomic formula into the form* $t_1 \geq t_2$*, where* t_1 *is a variable or the constant* `0` *and* t_2 *a monadic linear polynomial.*
9. *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* $u_i \geq u_j + c$*. Check if the graph has a positive cycle.*

Figure 4: The `arith` algorithm

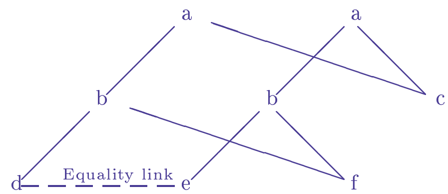
26.1.2 Equality Reasoning

Reasoning about equalities is the problem of verifying that one equality follows as a consequence of several other equalities, e.g. that $f(f(a,b),b) = a$ follows from $f(a,b) = a$ or that $g(a) = a$ follows from $ggg(a) = a$ and $ggggg(a) = a$. Nearly all the problems occurring in practice, particularly when reasoning about programs, require reasoning about equalities.

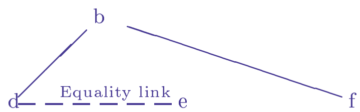
Equality reasoning essentially involves *reflexivity, symmetry, transitivity* and *substitution* rules and there are several algorithms for dealing with it. The one we will describe here is based on work by Greg Nelson and Derek. C. Oppen and has been successfully implemented as `equality` rule in PRL. Its key idea is constructing the *congruence closure* of a relation on a graph. This method is not restricted to equality reasoning but can be extended to reasoning about list structures and similar problems as well. Again, we will explain the algorithm by an example. Consider the goal

H, `d=e` $\vdash$ `a(b(d,f),c) = a(b(e,f),c)`

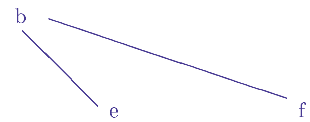
where `a, b, c, d, e, f` are arbitrary terms. We represent expressions as trees as usual and equalities of expressions as "equality links" between nodes. Thus proving two expressions to be equal is the same as showing that a link between the corresponding top nodes can be constructed. In the above formula the two expressions have `c` and `f` as common nodes while `d` and `e` are linked.



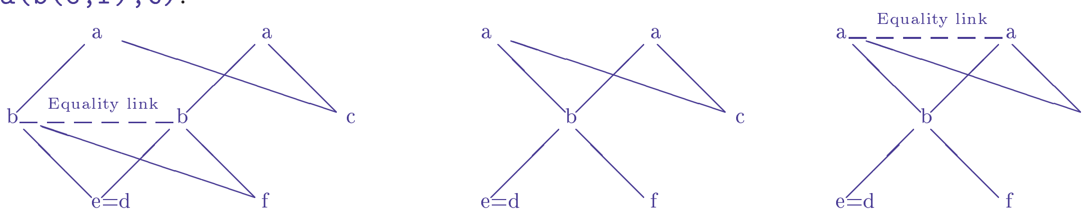
The equality link means that the nodes `d` and `e` can be lumped together which essentially is the same as changing



into



In the next step we look for subtrees which have `d` and `e` as subterms and are identical in the remaining nodes and construct an equality link between the top nodes of these trees. In our example we may link `b(d,f)` with `b(e,f)` and continue until we have linked `a(b(d,f),c)` with `a(b(e,f),c)`.



□

Mathematically, constructing links between nodes can be expressed as computing the congruence closure of a relation on a graph, which is defined as follows.

Let $G = (V, E)$ be a directed graph with labeled nodes (vertices). For each vertex v let $l(v)$ denote its label and $\delta(v)$ the number of edges leaving v. For $1 \leq i \leq \delta(v)$ let $v[i]$ denote the i-th successor of v. u is a *predecessor* of v if $v = u[i]$ for some i. Let R be a relation on V. Two vertices are *congruent under R* if $l(u) = l(v)$, $\delta(u) = \delta(v)$ and $(u[i], v[i]) \in R$ for all $1 \leq i \leq \delta(u)$. R is *closed under congruences* if for all vertices u and v which are congruent under R the relation $(u, v) \in R$ holds. The *congruence closure* of R is the unique minimal extension of R which is an equivalence relation and closed under congruences.

The algorithm *MERGE*, shown in Figure 5 constructs the congruence closure of $R \cup \{(u, v)\}$ where R is an arbitrary relation on a graph G which is closed under congruences and u and

---

Given a relation R on a graph G which is closed under congruences and two vertices u and v in G.

1. *If the equivalence classes of u and v are identical then return.*
2. *Let P_u be the set of all predecessors of vertices equivalent to u and P_v the set of all predecessors of vertices equivalent to v.*
3. *Combine the equivalence classes of u and v by a link.*
4. *For each pair (x, y) with $x \in P_u, y \in P_v$ do the following: If the equivalence classes of x and y are different and x and y are congruent then* MERGE *x with y.*

Figure 5: The *MERGE* algorithm

Given hypotheses s_1=t_1, . . . , s_n=t_n and a conclusion s=t,

1. *Construct a graph G that corresponds to the set of all terms in the hypotheses and the conclusion. Let $\tau(u)$ be the vertex in G representing u and R be the identity relation on the vertices of G.*
2. MERGE *$\tau(s_i)$ with $\tau(t_i)$ for all $1 \leq i \leq n$*
3. *If $\tau(s)$ is equivalent to $\tau(t)$ then s=t follows from the hypotheses. Otherwise it does not follow.*

Figure 6: The `equality` algorithm

v are vertices of G. For the sake of efficiency an equivalence relation is represented by the corresponding set of equivalence classes.

The following example of computing the shows how *MERGE* operates. We want to compute the closure of $ggggg(a) = a$ and $ggg(a) = a$.

It suffices to construct a graph for $ggggg(a)$ as on the right. Let R be (v_1, v_6) $(ggggg(a) = a)$, u=v_3, v=v_6. v_2 and v_5 are the predecessors of u and v. Combining the equivalence classes of u and v constructs the class $\{v_1, v_3, v_6\}$. Since the equivalence classes of the successors of v_2 and v_5 (that is of u and v) are now identical *MERGE* is called with v_2 and v_5. In the next step v_1 and v_4 are investigated while $\{v_2, v_5\}$ are combined. In the following step the combination of v_1 and v_4 yields $\{v_1, v_3, v_4, v_6\}$ and v_3 is checked to be congruent with v_2 (and later v_5). This calls *MERGE* on (v_3,v_2) which constructs the equivalence class $\{v_1, v_2, v_3, v_4, v_5, v_6\}$. Since all equivalence classes are now identical the algorithm stops. All six vertices are equivalent.

$$\begin{tikzcd}[row sep=small]
g \; (v_1) \arrow[d] \\
g \; (v_2) \arrow[d] \\
g \; (v_3) \arrow[d] \\
g \; (v_4) \arrow[d] \\
g \; (v_5) \arrow[d] \\
a \; (v_6)
\end{tikzcd}$$

Quantifier free equality reasoning with uninterpreted function symbols can be easily reduced to the congruence closure problem. To find out whether s=t follows from hypotheses s_1=t_1, . . . , s_n=t_n it suffices to stepwisely build the congruence closure of the given equalities and then find out whether s and t are equivalent. The algorithm is described in Figure 6.

26.2 Tactical Theorem Proving

Automating formal reasoning by adding decision procedures to the logic leads to a powerful reasoning mechanism, since there are even methods for making decision procedures cooperate. However, this approach does not generalize very well. When the formal logic becomes more and more expressive, it becomes increasingly more difficult to add procedures that can decide a significant part of the logic.

In particular, they are of little help for users of the formal system who want to define new concepts (like reasoning about graphs and trees) in terms of existing ones. While the definition mechanism allows them to use terminology tailored to formalizing the new concepts, the reasoning mechanism remains fixed, unless a system programmer adds the necessary special-purpose procedures to the proof environment. This, however, makes extensions to the reasoning apparatus impractical and bears the risk of introducing inconsistencies to the system. Therefore, it is necessary to provide a more flexible mechanism for extending the reasoning capabilities of a formal proof system while offering the greatest possible security against faulty proofs.

This idea, which first came up in the Edinburgh LCF Project, has been realized by allowing the user to *program proof strategies* interactively by accessing the *metalanguage* of the proof system in a controlled fashion. Users are given the freedom to analyze and build terms, formulas, sequents, and proofs, to *plan* proofs in advance, and to experiment with strategies that search for proofs. But they are prevented from manipulating proofs by other means than executing the primitive inference rules.

Proof methods that proceed by programming the application of elementary inference steps (usually called *tactics*) can be used to search for proofs, to modify existing ones, to hide uninteresting details, to structure complex proofs, and - together with the definition mechanism, to extend the formal language of type theory to user-defined concepts. The paradigm of *tactical theorem proving* provides a method to combine the accuracy of a machine with the ingenuity of human professionals which can turn a simple proof-checker into a powerful tool for proving theorems, generating programs, and developing formal mathematical theories.

The PRL systems have adopted Edinburgh's metalanguage ML as formalization of their metalanguage. ML is a fairly mathematical functional programming language that provides features like abstract data types, exception handling, and a rich type structure.[9] In λ-PRL's version of ML special datatypes can be used to refer to the object theory. The type `term` represents terms of the object language of λ-PRL and provides functions to access the subterms of a term and to build new terms from components. The recursive data type `proof` describes the structure of PRL's proof trees ad provides functions for accessing hypotheses, conclusion, and subgoals of the current goal.

26.2.1 Writing tactics

All proof rules of λ-PRL are implemented as tactics, i.e. as functions in `proof -> proof` and connected to a proof editor that executes them in place, i.e. replaces a proof by the result of applying a tactic to it. Further proof tactics can be implemented in terms of the elementary proof rules. The following tactics are predefined.

`Id`: is a tactic that does not change the goal.

`hypothesis` is a tactic which tries to find a hypothesis that proves the goal. One does not have to provide a hypothesis number anymore.

`false_elimination`: a tactic checking for inconsistent hypotheses

`and_elimination`: splitting all conjunctions in the hypotheses

`or_elimination`: splitting all disjunctions in the hypotheses

`implication_elimination`: executing `impL` on the first applicable hypothesis

[9] The modern programming languages SML and OCaml are derived from the original formulation of ML but deviate from its syntax and add a variety of new features that didn't exist in the early 80's.

t_1 **THEN** t_2: *apply t_2 to all the subgoals resulting from t_1.*

t **THENL** $[t_1; t_2; ..t_n]$: *apply t_i to the i_{th} subgoal resulting from t.*

t_1 **ORELSE** t_2: *apply t_1; if it fails apply t_2.*

REPEAT t: *repeat tactic t until it fails.*

COMPLETE t: *apply tactic t if it completes the proof.*

PROGRESS t: *apply tactic t if it makes progress.*

TRY t: *apply t; if it fails apply* `Id`

Figure 7: Important predefined tacticals

`cases:` combines the cut rule with a case split Instead of

$H \vdash T$ `BY cut` $A \vee B$
`1.` $H \vdash A \vee B$
`2.` H, $j{:}\, A \vee B \vdash T$ `BY orE` j
`2.1.` H, $A \vdash T$
`2.2.` H, $B \vdash T$

one may simply use

$H \vdash T$ `BY cases` $A \vee B$
`1.` $H \vdash A \vee B$
`2.` H, $A \vdash T$
`3.` H, $B \vdash T$

To combine simple tactics like the above into more sophisticated ones a user does not have to go into the details of ML. Instead, he may make use of tools called *tacticals*, which are defined as ML functions that map tactics into tactics. Figure 7 lists a few predefined ones which have proven particularly helpful for creating tactics.

Just with these tacticals, the λ-PRL rules, and predefined tactics one can already write remarkably powerful tactics. A very interesting one of this kind is a tactic performing "immediate" reasoning which is extremely helpful if one does not want to do all the boring proofs of easy facts. Its construction also shows how to set priorities on the tactics applied in a heuristic mechanism. The priorities in this case are calculated through *sorting by the costs of their application.*

```
let immediate =
 REPEAT
   (        hypothesis
    ORELSE  false_elimination
    ORELSE  arith
    ORELSE  equality
    ORELSE  and_elimination
    ORELSE  impR
    ORELSE  notR
    ORELSE  allR
    ORELSE  andR
    ORELSE  or_elimination
    ORELSE  implication_elimination
    )
;;
```

tac `...` $\equiv$ *tac* `THEN immediate`

```
⊢ ∀n:int.  0≤n ⊃ ∃y:int.  y^2≤n  ∧  n<(y+1)^2          BY induction ...
|\
| 0≤0 ⊢ ∃y:int.  y^2≤0  ∧  0<(y+1)^2                   BY exR 0 ...  √
 \
  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 cut n<(y+1)^2 ∨ n=(y+1)^2...
  |\
  | ..., n<(y+1)^2 ⊢ ∃y:int. y^2≤n ∧ n<(y+1)^2         BY exR y  ...  √
   \
    ..., n=(y+1)^2 ⊢ ∃y:int. y^2≤n ∧ n<(y+1)^2         BY exR y+1 ... √
```

Figure 8: Proof of the Integer Square Root Problem with Tactics

The first four tactics either fail or succeed without leaving subgoals and may therefore be considered cheap. The last one has relatively high costs because it could go wrong directions in a proof. Tactics like quantifier-elimination could have been included but are not in this particular tactics.

Figure 8 demonstrates the advantage of using tactics over primitive inference rules. Just with the tactics discussed in this section the proof of the Integer Square Root Problem can be significantly shortened to 6 basic steps, which describe its essential ideas, while the tedious and less interesting details are handled by the tactic `immediate`, which is executed after each step.

26.2.2 Validity

There is an open end to the ideas that a user may apply when programming tactics. Users are free to define their own reasoning system as long as it can be expressed in terms of the existing rules. This flexibility has turned out to be an especially successful feature in systems that are based on logics much richer than the logic of λ-PRL.

An important aspect of tactics is that one does *not have to worry about the correctness* of user-defined tactics. Tactics may fail to terminate, if applied carelessly, but because they can only use the inference rules of the underlying calculus they *always produce valid proofs*.