Technical Report

UCAM-CL-TR-67
ISSN 1476-2986

Number 67

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Natural deduction theorem proving via higher-order resolution

Lawrence C. Paulson

May 1985

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Natural Deduction Theorem Proving via Higher-Order Resolution

Lawrence C Paulson
Computer Laboratory
University of Cambridge
May 20, 1985

An experimental theorem prover is described. Like LCF, it is embedded in the metalanguage ML and supports backwards proof using tactics and tacticals. The prover allows a wide class of logics to be introduced using Church's representation of quantifiers in the typed lambda-calculus. The inference rules are expressed as a set of generalized Horn clauses containing higher-order variables. Depth-first subgoaling along inference rules is essentially linear resolution, but using higher-order unification instead of first-order. This constitutes a higher-order Prolog interpreter.

The rules of Martin-Löf's Constructive Type Theory have been entered into the prover. Special tactics inspect a goal and decide which Type Theory rules may be appropriate, avoiding excessive backtracking. These tactics can automatically derive the types of many Type Theory expressions. Simple functions can be derived interactively.

CONTENTS

1. The LCF interactive theorem prover

The LCF approach to theorem proving uses a natural deduction logic embedded in a programmable metalanguage (ML) [5]. Terms and formulas of the logic are values in ML: they have an explicit tree structure and can be decomposed and built up by ML functions. Theorems are values of an abstract ML type: there are no arbitrary constructors for theorems. *Inference rules* are functions from theorems to theorems, while *axioms* are special theorems recorded by LCF. Theorems can only be created by applying inference rules to axioms and other theorems. A derived rule is implemented by composing inference rules into a new ML function. Since both derived and primitive inference rules are ML functions, both can be used in the same way. But a derived rule can be inefficient. Each time it is called, it executes all the primitive inferences justifying it.

Most proofs are conducted backwards [24]. A *goal* is a representation of a desired theorem; a *tactic* is a function that reduces a goal to a list of subgoals. In LCF, this top-down decomposition of the goal must be followed by a bottom-up reconstruction of the proof. Tactics provide high level assistance in the search for a proof, but a theorem can only be produced by executing the primitive inference rules. Therefore each tactic returns a *validation*, a function that produces a theorem achieving the goal, given theorems achieving the subgoals. LCF composes the validations from the tactic steps in the proof.

Past LCF proofs have involved denotational semantics, verification of functional programs, and verification of digital circuits [21]. Various logics have been implemented: two versions of PPLAMBDA (for denotational semantics), a Logic for Sequential Machines, a higher order logic, and Martin-Löf's Constructive Type Theory. Implementing a logic is a major undertaking: choosing a representation of formulas, implementing several dozen inference rules and tactics, implementing many more derived rules and higher level tools, implementing a theory database, writing a parser and printer, and documenting all these things. Each rule checks that it has received suitable premisses, then generates the conclusion. Each tactic checks that it has received a suitable goal, then generates the subgoals giving the corresponding rule as validation. This uniform style of writing functions calls for automation.

2. Quantifiers and higher-order variables

The rules of natural deduction are usually presented as pattern transformations. Patterns involve syntactic metavariables ranging over the various constructs of the logic. For instance, if H ranges over lists of hypotheses, and A and B are formulas, then $\wedge$-introduction is often expressed:

$$\frac{H \vdash A \qquad H \vdash B}{H \vdash A \wedge B}$$

The discharging of assumptions is easily expressed, as in the $\Rightarrow$-introduction rule:

$$\frac{H, A \vdash B}{H \vdash A \Rightarrow B}$$

Rules that involve variables are troublesome. Let the metavariables x, y range over (object) variables. Consider $\forall$-introduction:

$$\frac{H \vdash A[x]}{H \vdash \forall y.A[y]} \qquad x \text{ not free in } H \tag{1}$$

The restriction on x prevents contradictions such as

$$\begin{aligned} & x = 0 \vdash x = 0 \\ & x = 0 \vdash \forall y.y = 0 && \text{falsely!} \\ & \vdash x = 0 \Rightarrow \forall y.y = 0 \\ & \vdash \forall z.z = 0 \Rightarrow \forall y.y = 0 \\ & \vdash \forall y.y = 0 && \text{contradiction!} \end{aligned}$$

Note that x is the universal variable in two distinct formulas, $x = 0$ and $x = 0 \Rightarrow \forall y.y = 0$.

How can we formalize the $\forall$-introduction rule (1), with its English restriction on x and the substitution implicit in the $[\cdots]$ notation? Higher-order variables come to the rescue, using Church's type theory — a representation of logic in the typed λ-calculus. Let **term** be the type of terms, and **form** the type of formulas. The binary connectives like conjunction and implication are easily handled: let $\wedge$ and $\Rightarrow$ have type **form** $\to$ (**form** $\to$ **form**). If $A \in$ **form** and $B \in$ **form**, then also $A \wedge B \in$ **form**.

To handle quantifiers, introduce the symbol Π of type (**term** $\to$ **form**) $\to$ **form**. Let B be a function from terms to formulas: $B \in$ **term** $\to$ **form**. Then

$B(t) \in \mathbf{form}$ for all $t \in \mathbf{term}$, and $\Pi(B)$ represents the formula $\forall y.B(y)$. If $A[y]$ is a formula with free variable y, then $\lambda y.A[y]$ has type $\mathbf{term} \to \mathbf{form}$ and $\Pi(\lambda y.A[y])$ represents $\forall y.A[y]$. The sole binding operator is λ. The rule for $\forall$-introduction becomes

$$\frac{H \vdash B(x)}{H \vdash \Pi(B)} \qquad x \text{ not free in } H, B \tag{2}$$

Remember to read $\Pi(B)$ as $\forall y.B(y)$. There is a new restriction on x: it may not appear in B. The dependence of $B(x)$ upon x is purely one of function application. The $[\cdots]$ notation has become $(\cdots)$; β-conversion handles all questions of substitution. The corresponding $\forall$-elimination rule is simply

$$\frac{H \vdash \Pi(B)}{H \vdash B(t)}$$

The $\forall$-introduction rule (2) is not yet ready to be automated. It contains the English restriction that the variable x must not appear in B or H. Apart from this restriction the variable name has no significance. So we might as well reformulate the rule to insist upon a particular name that cannot possibly appear in B or H. Extend the object syntax with a family of constant symbols $\mathbf{all}_{H,B}$ for all hypotheses H and formulas B. Now symbols and expressions are mutually recursive: H and B are part of the symbol $\mathbf{all}_{H,B}$. Clearly $\mathbf{all}_{H,B}$ cannot appear in H or B since expressions are finitely constructed.

Using the new constants, $\forall$-introduction becomes

$$\frac{H \vdash B(\mathbf{all}_{H,B})}{H \vdash \Pi(B)} \tag{3}$$

Compare with Schmidt's natural deduction rules for Skolemization [24]. He tags a Skolem constant with B's free variables rather than with B itself. Suppes suggests similar rules for quantifiers, with helpful discussion [27] (pages 80–100). Existential quantifiers can be handled as negated universals, or with another constant Σ of type $(\mathbf{term} \to \mathbf{form}) \to \mathbf{form}$. Other rules involving bound variables, such as $\exists$-elimination, are Skolemized in the same way.

Perhaps the hypotheses H should not be a subscript of **all**. There is nothing similar in the literature on Skolemization. Rules like (3) slow down my implementation because H often gets bound to a large expression. The simpler

$\forall$-introduction rule

$$\frac{H \vdash B(\mathbf{all}_B)}{H \vdash \Pi(B)}$$

resembles Robinson's [23]. His logic includes *exemplification terms*, a version of Hilbert's ϵ-operator. The rule allows $\mathbf{all}_B$ to appear in the hypotheses H. Its soundness is clear if we only accept models that assign $\mathbf{all}_B$ a value y, if such exists, to make $B(y)$ false. In such a model, if $B(\mathbf{all}_B)$ is true then so is $\Pi(B)$.

Thus the family of symbols $\mathbf{all}_{H,B}$ can be understood *syntactically* as a way of choosing a name, or *semantically* as a choice function. One reason for using the subscripted symbol $\mathbf{all}_{H,B}$ instead of the function application $\mathbf{all}(H, B)$ is to avoid making **all** explictly a function in the object language. The choice function **all** may be unacceptable in a constructive logic. The symbol $\mathbf{all}_{H,B}$ is atomic, while $\mathbf{all}(H, B)$ contains well-formed subexpressions **all** and $\mathbf{all}(H)$. I tried the $\mathbf{all}(H, B)$ kind of Skolemization: the numerous subexpressions were an intolerable burden on the implementation of higher-order unification.

3. Representing logics in the typed λ-calculus

The previous discussion suggests a practical computer representation of formal logics. Logical expressions are represented in the typed λ-calculus, built up from constants, free and bound variables, abstractions, and combinations. We also have infinitely many Skolem constants subscripted by λ-expressions.

Each term has a type, denoted by Greek letters $\alpha, \beta, \ldots$. Types are recursively formed: there is a set of *ground* types, and also *function* types $\alpha \to \beta$.

- Each variable and constant has a type.
- If x has type α and t has type β then the abstraction $\lambda x.t$ has type $\alpha \to \beta$.
- If t has type $\alpha \to \beta$ and u has type α then the term $t(u)$ has type β.

There is a ground type for each syntactic class: terms, formulas, hypotheses, sequents, etc. A variable of type **term** ranges over terms of the logic.

A logic consists of *inference rules* of the form

$$\frac{P_1 \quad \cdots \quad P_m}{Q}$$

A rule without premisses ($m = 0$) is a *theorem*. There are two meta-rules, which produce new rules:

1. Any *instance* of a rule is a rule: any free variable may be substituted by an expression of the same type.

2. The *composition* of two rules is a rule: if

$$\frac{P_1 \quad \cdots \quad P_m}{Q_i} \qquad \frac{Q_1 \quad \cdots \quad Q_n}{R},$$

are rules ($1 \leq i \leq n$), then so is

$$\frac{Q_1 \quad Q_{i-1} \quad P_1 \quad \cdots \quad P_m \quad Q_{i+1} \cdots \quad Q_n}{R}.$$

Natural deduction is easily represented, though it is *not* built in. In the logic of the previous section, $\vdash$ is a infix constant of type **hyp** $\rightarrow$ (**form** $\rightarrow$ **sequent**).

In LCF the fundamental abstract type is *theorem*; here the fundamental abstract type is *rule*. LCF rules are functions; here rules are data structures. Composition of rules buids proof trees, with forwards and backwards proof as special cases. More advanced proof methods can be implemented. The *resolution* method is to *unify* the conclusion of one rule with a premiss of another, composing the corresponding instances of the rules. Theorem-proving tools employing unification have been used in some LCF proofs. Unification is particularly difficult when higher-order variables are present. These issues are discussed below.

4. LCF tactics using unification

Brian Monahan implemented several resolution tactics for Edinburgh LCF [18]. These work on the current goal's assumption list, adding resolvents as new assumptions. Previous resolution tactics used one-way matching; Monahan implemented proper unification. Monahan also automated the construction of simple inference rules and tactics. His function METARULE turns any theorem $P_1 \wedge \ldots \wedge P_m \Rightarrow Q$ into the rule $\frac{P_1 \quad \cdots \quad P_m}{Q}$. His METATAC produces the corresponding tactic. Their generality is limited because LCF's logic, PPLAMBDA, does not have variables ranging over formulas.

Sokołowski used Edinburgh LCF to prove the soundness of Hoare axiomatic rules with respect to a denotational definition of a simple programming language

[26]. The proof consists mainly of the systematic expansion of definitions. LCF's simplifier expands definitions by rewriting, but Sokołowski decided to structure his proof in terms of derived inference rules rather than rewrite rules. He implemented his own tools for depth-first search using backwards chaining along inference rules.

Previously each LCF goal had to be expressed in full. Sokołowski's innovation was to allow *pattern variables* in goals, and allow tactics to instantiate pattern variables by unification. Existential goals are the most obvious use for pattern variables: to prove $\exists x.P(x)$ it suffices to prove $P(t)$, for any term t. LCF and other theorem provers require the user to supply this *existential witness* right away. Sokołowski's tactics allow t to be inferred later in the proof. (Sokołowski was using Edinburgh LCF, which lacks existential quantifiers. The same reasoning holds for universal quantifiers in assumptions.)

This approach requires an *environment* to contain values for the pattern variables as they are discovered. A unification tactic takes an environment as well as a goal. It returns an extended environment along with the subgoals and validation. After all subgoals have been solved, the validation must be given the final environment as well as a list of theorems. Sokołowski implemented unification tactics and simple backtracking tactics in ML. His code ran slowly because LCF provided little support for unification. His treatment of environments during sequential composition (the tactical THEN) could be criticised. Despite these faults, Sokołowski's tactics verified the Hoare rules with remarkable clarity, capturing the high-level structure of the proofs.

5. Higher-order unification

Unifying two terms t and u means solving the equation $t = u$, determining values for the variables of t and u. For ordinary unification, terms are recursively built up from variables x and function applications $\mathbf{F}(t_1, \ldots, t_q)$. (Boldface letters denote constants.) Two terms are equal only if they have exactly the same structure above their variables. Typically terms and variables are untyped. It is decidable whether two terms can be unified; if they can be, then a most general unifier can be efficiently computed [12].

Higher-order unification amounts to solving equations in the typed λ-calculus

[7,9]. Two terms are equal if they can be made identical by some sequence of α, β, and η conversions:

α renaming of bound variable, $\lambda x.t = \lambda y.t[y/x]$

β substitution of argument into function body, $(\lambda x.t)u = t[u/x]$

η extensionality of functions, $\lambda x.t(x) = t$ if x is not free in the function t

Let $t(u_1, \ldots, u_q)$ abbreviate $(\ldots(t(u_1))\ldots)(u_q)$. As a subterm of some larger term, $t(u_1, \ldots, u_q)$ is *rigid* if t is a constant or bound variable, and *flexible* if t is a free variable.

Higher-order unification is *semi-decidable*: if the terms cannot be unified, the search for unifiers may diverge. Although a complete set of unifiers can be recursively enumerated, it may be infinite. Unifying the term $f(x)$ with the constant $\mathbf{A}$ gives the two unifiers $\{f = \lambda x.\mathbf{A}\}$ and $\{f = \lambda x.x,\ x = \mathbf{A}\}$. Unifying $f(x)$ with $\mathbf{A}(\mathbf{B}_1, \ldots, \mathbf{B}_q)$ gives $q + 2$ unifiers:

$$\{f = \lambda x.\mathbf{A}(\mathbf{B}_1, \ldots, \mathbf{B}_q)\}$$
$$\{f = \lambda x.x, \quad x = \mathbf{A}(\mathbf{B}_1, \ldots, \mathbf{B}_q)\}$$
$$\{f = \lambda x.\mathbf{A}(\mathbf{B}_1, \ldots, \mathbf{B}_{i-1}, x, \mathbf{B}_{i+1}, \ldots, \mathbf{B}_q), \quad x = \mathbf{B}_i\} \qquad i = 1, \ldots, q$$

Unifying $f(x_1, x_2)$ with $\mathbf{A}(\mathbf{B}_1, \ldots, \mathbf{B}_q)$ gives $q^2 + q + 3$ unifiers. The search space can easily become unmanagable.

Sometimes both terms begin with a function variable. Unifying $f(t_1, \ldots, t_p)$ with $g(u_1, \ldots, u_q)$ yields an explosion of counterintuitive solutions [9]. Huet calls this the *flex-flex* case. It always has the trivial solution

$$\{f = \lambda x_1 \ldots x_p.h,\ g = \lambda y_1 \ldots y_q.h\}\ .$$

Most implementations of higher-order unification use Huet's search algorithm, with procedures SIMPL and MATCH [7]. SIMPL essentially does first-order unification. Each term is put into normal form. Outermost lambdas are stripped off; the bound variables become part of the context, behaving much like constants. The input pair of terms is broken into a set of *disagreement pairs* to be unified. A *rigid-rigid* pair $\langle \mathbf{F}(t_1, \ldots, t_q), \mathbf{F}(u_1, \ldots, u_q) \rangle$ is simplified to the set of pairs $\langle t_1, u_1 \rangle, \ldots, \langle t_q, u_q \rangle$. If $\mathbf{F} \neq \mathbf{G}$ then $\langle \mathbf{F}(t_1, \ldots, t_p), \mathbf{G}(u_1, \ldots, u_q) \rangle$ is recognised as non-unifiable. A pair $\langle x, t \rangle$ has the most-general unifier $\{x = t\}$ if x does not

occur in t. The occur-check is subtle: x and $f(x)$ are unified by both $\{f = \lambda y.x\}$ and $\{f = \lambda y.y\}$. Huet's *rigid path* occur-check gives a practical sufficient condition for x and t to be unifiable.

MATCH guesses instantiations of function variables. A *flex-rigid* pair

$$\langle f(t_1, \ldots, t_p), \mathbf{F}(u_1, \ldots, u_q) \rangle$$

gives rise to as many as $p + 1$ possible substitutions for f. For $i = 1, 2, \ldots$, let h_i be a new variable of appropriate type, and let v_i denote the term $h_i(x_1, \ldots, x_p)$. We have

$$f = \lambda x_1 \ldots x_p.\mathbf{F}(v_1, \ldots, v_q), \quad \text{by } \textit{imitation};$$
$$f = \lambda x_1 \ldots x_p.x_i(v_1, \ldots, v_m), \quad \text{for certain } i = 1, \ldots, p, \text{ by } \textit{projection}.$$

The projection rule applies when the type of x_i allows m to be chosen to give f the correct type. Formally, if the type of f is $\alpha_1 \to \cdots \to \alpha_p \to \beta$, then α_i must be $\gamma_1 \to \cdots \to \gamma_m \to \beta$. Huet also has an algorithm for unification without η-conversions, when many more independent substitutions are possible.

This choice of substitutions creates an OR tree, most simply searched in depth-first order. One of these substitutions is chosen, applied to all the disagreement pairs, and the search continues. Many of the substitutions from the projection rule are immediately rejected if t_i begins with a constant different from $\mathbf{F}$. The imitation rule reduces the original disagreement pair to the pairs

$$\langle h_1(t_1, \ldots, t_p), u_1 \rangle, \quad \ldots, \quad \langle h_q(t_1, \ldots, t_p), u_q \rangle.$$

Huet's algorithm reports success when only flex-flex pairs remain. A theorem prover can store these as constraints on future unifications, or produce the trivial solution.

Of course theorem proving is undecidable, but it is unfortunate that our basic inference step is also undecidable. Other difficult unification problems can arise. For example, $\wedge$-introduction could be expressed to form the union of the hypotheses:

$$\frac{H_1 \vdash A_1 \qquad H_2 \vdash A_2}{H_1 \cup H_2 \vdash A_1 \wedge A_2}$$

Now unification must also handle $\cup$, an associative, commutative, and idempotent operator. To avoid this complexity the logic must be reformulated.

6. Higher-order theorem provers

The earliest applications of higher-order unification extended resolution to higher-order logic [9]. Huet's *constrained resolution* postpones branching in unification [8]. Rather than returning multiple unifiers in a resolution step, it records the remaining disagreement pairs as a *constraint* on the new clause. Further resolutions may satisfy the constraints or render them clearly unsatisfiable.

The TPS theorem prover uses sophisticated heuristics in the search for higher-order unifiers [15]. In MATCH it chooses a disagreement pair likely to cause the least branching of the tree. It hashes disagreement sets to determine whether a new set is subsumed by an older one. Though the subsumption test is expensive it cuts the search space substantially and prevents some searches from diverging. TPS uses *general matings* rather than resolution. The mating approach unifies subformulas against each other without reducing everything to clause form. TPS can automatically prove Cantor's Theorem, that every set has more subsets than elements [1].

The EKL proof checker uses higher-order matching of rewrite rules [11]. The AUTOMATH project has investigated several higher-order λ-calculi, reminiscent of Martin-Löf's type theory, as languages for machine-checked proof [10]. Gordon's HOL is a version of LCF for proving theorems in Church's higher-order logic [6]. The logics of HOL, EKL, and TPS are all descended from Church's.

Although several of these theorem provers support natural deduction, none use the higher-order representation of inference rules proposed here. The resolution-based systems use resolution as the only inference rule. My proposal may be seen as a two-level system of logic with resolution of inference rules.

7. A preliminary implementation

The theorem prover consists of about 2200 lines of Standard ML [17]. The compiler is David Matthew's Poly/ML, developed on a Cambridge VAX/750 computer running Berkeley Unix. The prover includes

- a data structure for terms and types, with simple utility functions
- a naive implementation of Huet's higher-order unification algorithm

- tactics for using higher-order Horn clauses in backwards proof
- a collection of functions for interactively growing a tree of goals
- the rules of Martin-Löf's type theory, a special printer, and special tactics.

Only the last of these items, about 25% of the code, is specific to Martin-Löf's logic. Variables (for unification) are represented by integers. Each bound variable is also represented by an integer, referring to the depth at which it is bound [3]. The unification function returns a possibly infinite stream of unifiers. Streams are an abstract type implemented as usual: each member contains a function for computing the rest of the stream.

Tactical proof is now an AND/OR tree, not an AND tree like in LCF [24]. Each tactic may return a stream of decompositions of the goal. Each decomposition is a rule $\frac{P_1 \cdots P_m}{Q}$, stating that the goal Q can be reduced to the goals $P_1, \ldots, P_m$. No validation function is needed, since the key objects are rules rather than theorems.

The tactic *RulesTac* takes a collection of rules and unifies the conclusion of each rule against the goal. There are two levels of choice: several rules may apply; a rule may have several unifiers with the goal. The tactical DEPTH_FIRST applies a tactic in depth-first search to completely solve a goal. If the tactic returns an empty stream of decompositions, depth-first search cannot proceed and the goal is abandoned. Sokołowski used depth-first search for backwards chaining along hypotheses [26]. I find it effective with introduction rules. Together with *RulesTac* it constitutes a higher-order Prolog interpreter. It can execute simple Prolog programs (slowly).

The unification function spends a lot of time in the occur-check. Before binding a variable x to a term t, it must scan t for occurrences of x. The goals with their assumption lists are represented by ever-growing terms that are scanned repeatedly. Most Prolog compilers omit the occur-check, but it is essential here to enforce the formalized constraints "x not free in H, B." I have achieved reasonable efficiency by simple means. Ideas from the fast first-order unification algorithms might allow a speedup.

The original goal may contain variables representing "don't care" values or information to be inferred. Other variables crop up as existential witnesses. Many variables appear in more than one goal. When a unification tactic instantiates a variable, the environment effectively substitutes the new value in the other goals.

The existential witnesses develop step by step. To minimize backtracking, goals must be tackled in a sensible order, propagating constraints to other goals. Search is often constrained by the structure of the goal; if the goal has a variable in a critical place, the search becomes unconstrained. A goal is *too flexible* if certain of its variables give too many choices for the next step of the search.

I am experimenting with a depth-first tactical that delays goals that are too flexible. DEFLEX repeatedly applies a tactic to a goal and its subgoals. The tactic can reject a goal as too flexible by raising an exception, giving some of the goal's free variables. A rejected goal is not abandoned but queued, along with the free variables. The queued goals are re-examined after all the other goals have been solved. If any of the free variables have become instantiated, the goal can be reconsidered. Goals that are still too flexible are returned as subgoals. Goals can also be rejected as inappropriate for the particular tactic. This allows a layered approach: first attack all goals of one kind, then clean up the remaining goals.

The interactive goal package maintains a tree of subgoals. The user can apply a tactic to any number of the goals. A goal is called *closed* when first created, and *open* when a tactic has reduced it to zero or more subgoals. When applying a tactic, the package uses the first set of subgoals produced, saving the remainder of the stream. The user can explicitly *backtrack* any open goal. This takes the next set of subgoals, discarding the previous set along with any following proof steps. The backtrack command fails if that goal has no more subgoal decompositions to try. The package does not record the most recent choice point for backtracking.

8. Implementing Constructive Type Theory

Martin-Löf's Type Theory is an attempt to formalize constructive reasoning [19,20]. The rules for each logical connective express its constructive interpretation as operations on proof objects, giving interpretation of *propositions as types*. For instance, a proof of $A \wedge B$ is a pair $\langle a, b \rangle$, where a is a proof of A and b is a proof of B. The proof objects for the connectives constitute a simple functional programming language. It can express all provably terminating computations. "Propositions as types" allows a small family of primitives to provide a full system of logical connectives, data structures, and programs. A data type can express a complete

formal specification. The type of a sorting function can assert that its output is a sorted permutation of its input.

Constructive Type Theory has several kinds of rules:

- *Formation* rules build types from other types.
- *Introduction* rules build elements of types.
- *Elimination* rules specify control structures, called *selectors*, for each type.
- *Equality* rules give the result of evaluating expressions.

There are four kinds of judgement (theorem):

- A **type** means that A is a type.
- $A = B$ means that A and B are equal types.
- $a \in A$ means that a is an element of type A.
- $a = b \in A$ means that a and b are equal elements of type A.

Judgements can be hypothetical. Hypotheses have the form $x_1 \in A_1$, ..., $x_n \in A_n$, where the A_i are well-formed types. Under these assumptions $x_i \in A_i$ is a valid judgement. Verifying this requires searching down the list of hypotheses to find x_i. The rules on the computer use the Prolog style of list processing:

$$\frac{H \vdash A \textbf{ type}}{H, a \in A \vdash a \in A} \qquad \frac{H \vdash a \in A}{H, b \in B \vdash a \in A}$$

One of the virtues of Prolog is that a rule can specify more than one direction of information flow: programs can run backwards. Proving Martin-Löf theorems via unification can solve different classes of problems. The judgement $a \in A$ has several meanings, including

- a is a program of type A,
- a is a program with specification A,
- a is a proof of the proposition A.

Beginning with a program a, proving $a \in A$ by unification determines the type of a remarkably as Milner's algorithm for *type inference* does [16]. Beginning with a specification A, proving $a \in A$ amounts to the *program synthesis* of a. Beginning with a proposition A, proving $a \in A$ gives a *constructive proof* of A .

Martin-Löf's system has attracted increasing attention from computer scientists interested in formal verification and derivation of programs. Petersson has

incorporated it into Edinburgh LCF, producing the Gothenburg Type Theory system [22]. Chisholm used this to derive a parser [4]. Constable and his colleagues are modifying LCF to implement a related type theory [2]. My original aim was to implement unification tactics for proving theorems in Martin-Löf's type theory. I now hope to handle a broad class of formal logics.

The rules are normally presented within a higher-order framework. Though published material is scarce [14], insiders know that these rules are based on a theory of expressions. Each expression has an *arity* written as a jumble of parentheses. Apart from the peculiar notation, these expressions are simply the typed λ-calculus with a single ground type. There is a two-level type system; arities are the lower-level types. Consider the similarity between the $\forall$-introduction rule and Martin-Löf's rule of Π-introduction:

$$\frac{H \vdash A\ \mathbf{type} \qquad H, x \in A \vdash b(x) \in B(x)}{H \vdash \lambda y.b(y) \in \Pi_{y \in A} B(y)} \qquad x \text{ not free in } H, b, B$$

This rule is already higher-order; we need only Skolemize x. My computer formulation uses a Skolem constant $\mathbf{pri}_{b,B}$ for **p**roduct **i**ntroduction. There is no subscript H (which I hope is all right):

$$\frac{H \vdash A\ \mathbf{type} \qquad H, \mathbf{pri}_{b,B} \in A \vdash b(\mathbf{pri}_{b,B}) \in B(\mathbf{pri}_{b,B})}{H \vdash \lambda(b) \in \Pi(A, B)}$$

The ML definition of this rule is

```
val ProdIntrRl =
  let val pri = skolem ("pri", [b1,B1]) in
  prepare_rule
    (elemG$$ [lambda$b1, Prod$A$B1] $ H  :-
        [ typeG $ A $ H,
          elemG$$ [b1$pri, B1$pri] $ ((pri,A) ::* H)])  end;
```

I have written tactics for solving problems in Type Theory. The simplest problem is to check that a type A is well-formed. This is solved by repeatedly using formation rules to prove the judgement A **type**. Another common problem is to construct some element of a type A. This is clearly undecidable: it amounts to proving the proposition represented by A. Yet repeated use of introduction rules can perform a large portion of the proof.

The tactic *FormTac* is simply a call of *RulesTac* giving all the formation rules. Likewise there is an *IntrTac*, etc. These are work badly with backtracking: if a variable is encountered an infinite stream of types may be generated. The tactic *FlexFormIntrTac* uses DEFLEX to handle variables sensibly in the formation and introduction rules. For the goal A **type**, it raises an exception unless A is rigid (not a variable). For $a \in A$ either a or A must be rigid.

The tactic *TypeCheckTac* is similar but uses elimination rules as well as formation and introduction rules. For $a \in A$ it requires a to be rigid, since if a is a variable then all the elimination rules apply. *TypeCheckTac* handles the type-checking problems that have come up in my experiments. For instance it can check the type of the addition operator. It solves the goal

$$\lambda(k)\lambda(m)\,\mathbf{rec}(m, (x, y)succ(y), k) \in A\,,$$

producing the binding $A = \mathbf{Nat} \to \mathbf{Nat} \to \mathbf{Nat}$.

The selectors are higher-order combinators. Even with syntactic sugar they make Type Theory programs tedious to write and difficult to read. I am interested in the interactive derivation of programs that meet specifications. The simplest kind of specification consists of equations, one for each possible input pattern (like ML function definitions). Experimental tactics for equality rules can handle some examples with little user guidance.

For the *predecessor function* the tactics discover that $pred = \lambda(x)\,\mathbf{rec}(x, (y, z)y, x)$ with type $T = \mathbf{Nat} \to \mathbf{Nat}$ by solving the goal

$$x \in \sum_{pred \in T} \left(pred[\![zero]\!] =_{\mathbf{Nat}} 0 \times \prod_{k \in \mathbf{Nat}} pred[\![\mathbf{succ}\ k]\!] =_{\mathbf{Nat}} k \right) .$$

For the projection function *fst*, the tactics discover that $fst = \lambda(x)\,\mathbf{split}((y, z)y, x)$ with type $T = (\mathbf{Nat} \times \mathbf{Nat}) \to \mathbf{Nat}$:

$$x \in \sum_{fst \in T} \prod_{i \in \mathbf{Nat}} \prod_{j \in \mathbf{Nat}} fst[\![\langle i, j \rangle]\!] =_{\mathbf{Nat}} i$$

In this example the tactics discover a binding for f involving the selectors **when** and **split**, and determine that $N = \mathbf{Nat}$:

$$x \in \prod_{i \in \mathbf{Nat}} \prod_{j \in \mathbf{Nat}} f(\mathbf{inl}\langle i, j \rangle) =_N i \times f(\mathbf{inr}\langle i, j \rangle) =_N j$$

Recursive functions are harder to handle. The simple **rec** form of arithmetic addition can be discovered. Deriving addition as a function of type **Nat** → **Nat** → **Nat** requires computation on both numbers and function types. The recursion equations must be simplified as the proof proceeds. Recursive disagreement pairs often cause the unification algorithm to diverge.

9. Future work

These results show that higher-order unification can support practical theorem proving. Much work remains before more interesting proofs can be attempted.

Skolemization has been the number one trouble spot. For each logic, the Skolemized forms of its rules must be justified. Constructive logics offer special challenges. Efficiency requires Skolem constants to have as few subscripts as possible. The unification algorithm must be extended to handle subscripts.

LCF's *simplifer* does most of the work in proofs. Using equations as rewrite rules, it reduces a goal to a simpler one. Some form of simplifier is essential, though the present framework favours a decomposition approach: one goal may be reduced to several.

Higher-order unification usually behaves well. Most of the time just first-order unification takes place. For this reason there is no need to hash disagreement sets and detect subsumption, while the TPS group found subsumption essential. Unification may loop if asked to solve recursion equations. Some examples cause the search space to explode.

LCF's *theory package* maintains a data base of constants, types, axioms, and theorems on disc. LCF provides no way of storing relevant tactics and rules with a theory. A theory is also a logical notion. The logic should include inference rules for constructing and combining theories.

The *user interface* is crude. In LCF the logic is integrated with the ML compiler, syntax and all. There is no proper way to extend the Standard ML parser and printer. Logical formulas must be written in abstract syntax; the printer must be explicitly invoked. Goals are designated by number. A high-resolution display and pointing device would make life much easier, even in this early stage of development.

The theorem prover is one of the first large programs written in Standard ML, and the first run on Matthew's ML compiler. My experiences with both the language and the compiler are positive. ML's type system catches many bugs. Even sophisticated code often works after the first successful compilation. Subtle bugs can crop up weeks later, though. Debugging is especially hard when the compiler itself may contain bugs. There were many compiler bugs at first, but Matthews quickly fixed most of them. Compiled code runs fast: examples involving dozens of higher-order unifications need only seconds of VAX computer time. A profiling command lists the time and storage used by each function. This makes it easy to tune the code.

Acknowledgements. David Matthews has put considerable effort into the Standard ML compiler written in his language Poly. I thank the SERC for funding this. I have discussed these ideas with many people, particularly those involved with TPS.

REFERENCES

[1] P. B. Andrews, D. A. Miller, E. L. Cohen, F. Pfenning, Automating higher-order logic, in: W. W. Bledsoe and D. W. Loveland, editors, *Automated Theorem Proving: After 25 Years*, American Mathematical Society (1984), pages 169–192.

[2] J. L. Bates, R. L. Constable, Proofs as programs, *ACM Transactions on Programming Languages and Systems* **7** (1985), pages 113-136.

[3] N. G. de Bruijn, Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem, *Indagationes Mathematicae* **34** (1972), pages 381–392.

[4] P. Chisholm, Derivation of a parsing algorithm in Martin-Löf's theory of types, Dept. of Computer Science, Heriot-Watt University, Edinburgh (1985).

[5] M. J. C. Gordon, R. Milner, and C. P. Wadsworth, *Edinburgh LCF: A Mechanised Logic of Computation*, Springer LNCS 78 (1979).

[6] M. J. C. Gordon, HOL: A machine oriented formulation of higher order logic, Computer Lab., University of Cambridge (1985).

[7] G. P. Huet, A unification algorithm for typed λ-calculus, *Theoretical Computer Science* **1** (1975), pages 27–57.

[8] G. P. Huet, A mechanization of type theory, *Third International Joint Conference on Artificial Intelligence* (1973).

[9] D. C. Jensen, T. Pietrzykowski, Mechanizing ω-order type theory through unification, *Theoretical Computer Science* **3** (1976), pages 123–171.

[10] L. S. van Benthem Jutting, *Checking Landau's 'Grundlagen' in the AUTOMATH system*, PhD Thesis, Technische Hogeschool, Eindhoven (1977).

[11] J. Ketonen, EKL— A mathematically oriented proof checker, in: R. E. Shostak, editor, *Seventh Conference on Automated Deduction*, Springer LNCS 170 (1984), pages 65–79.

[12] A. Martelli, U. Montanari, An efficient unification algorithm, *ACM Transactions on Programming Languages and Systems* **4** (1982), pages 258–282.

[13] P. Martin-Löf, Constructive mathematics and computer programming, in: L. J. Cohen, J. Los, H. Pfeiffer and K.-P. Podewski (editors), *Logic, Methodology, and Science VI*, North Holland (1982), pages 153–175.

[14] P. Martin-Löf, Intuitionistic type theory, Unpublished notes prepared by G. Sambin (1980).

[15] D. A. Miller, E. L. Cohen, P. B. Andrews, A look at TPS, in: D. W. Loveland, editor, *Sixth Conference on Automated Deduction*, Springer LNCS 138 (1982), pages 50–69.

[16] R. Milner, A theory of type polymorphism in programming, *Journal of Computer and System Sciences* **17** (1978).

[17] R. Milner, A proposal for Standard ML, *ACM Symposium on Lisp and Functional Programming* (1984), pages 184–197.

[18] B. Q. Monahan, *Data Type Proofs using Edinburgh LCF*, PhD Thesis, University of Edinburgh (1984).

[19] B. Nordström, Programming in Constructive Set Theory: some examples, *ACM Conference on Functional Programming Languages and Computer Architecture* (1981) pages 141–153.

[20] B. Nordström and J. Smith, Propositions and specifications of programs in Martin-Löf's type theory, *BIT* **24** (1984), pages 288–301.

[21] L. C. Paulson, Lessons learned from LCF: A Survey of Natural Deduction Proofs, *Computer Journal* **28** (1985), to appear.

[22] K. Petersson, A programming system for type theory, Report 21, Department of Computer Sciences, Chalmers University, Göteborg, Sweden (1982).

[23] J. A. Robinson, *Logic: Form and Function*, Edinburgh University Press, 1979.

[24] D. Schmidt, A programming notation for tactical reasoning, in: R. E. Shostak, editor, *Seventh Conference on Automated Deduction*, Springer LNCS 170 (1984), pages 445–459.

[25] S. Sokołowski, A note on tactics in LCF, Report CSR-140-83, Dept. of Computer Science, University of Edinburgh (1983).

[26] S. Sokołowski, An LCF proof of the soundness of Hoare's logic, Report CSR-146-83, Dept. of Computer Science, University of Edinburgh (1983), to appear in *ACM Transactions on Programming Languages and Systems.*

[27] P. Suppes, *Introduction to Logic*, van Nostrand (1957).