The use of machines to assist in rigorous proof

Rodrigo Raya

École Polytechnique Fédéral de Lausanne

May 7, 2017

Overview

1. The paper's history and its importance at the time
2. A theory of goal seeking
3. An example: a theorem about parsing
4. Posterior developments
5. Conclusion and references

Note: the dates employed in this document correspond to the publication of the relevant research papers.

1 The paper's history and its importance at the time

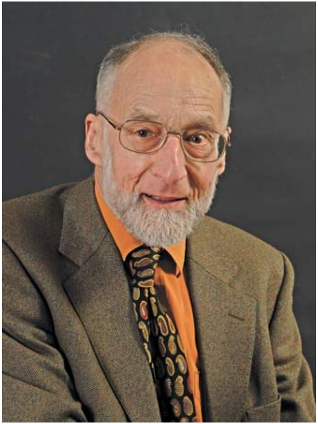
2 A theory of goal seeking

3 An example: a theorem about parsing

4 Posterior developments

5 Conclusion and references

Robin Milner (1934,2010)



ACM A.M. Turing Award (1991). For three distinct and complete achievements:

- LCF, the mechanization of Scott's Logic of Computable Functions, probably the first theoretically based yet practical tool for machine assisted proof construction.
- ML, the first language to include polymorphic type inference together with a type-safe exception-handling mechanism.
- CCS, a general theory of concurrency.

In addition, he formulated and strongly advanced full abstraction, the study of the relationship between operational and denotational semantics.

From Hoare logic to denotational semantics

At Swansea University (1968), Milner developed his first automatic theorem prover:

- Based on Robinson's resolution principle (1965) and Hoare-Floyd logic (1967-9).
- Program correctness generating verification conditions.
- Verification conditions stated as first-order logic formulae.
- The proof of verification conditions formulae with Robinson's resolution method took too long and limited the amount of theorems to proof.

From Hoare logic to denotational semantics

- Around 1970, Milner gets familiars with the works of Dana Scott and Christopher Strachey that founded the theory of denotational semantics.

> *"I could write down the syntax of a programming language in this logic and i could write the semantics in the logic"*

- The automated reasoning implementation of Scott's logic became Standford LCF (logic of computable functions)

Stanford logic of computable functions

- An interactive user-guided system.
- Programs could be reasoned about directly via their semantics encoded in Scott's logic.
- Based on a goal-directed reasoning.

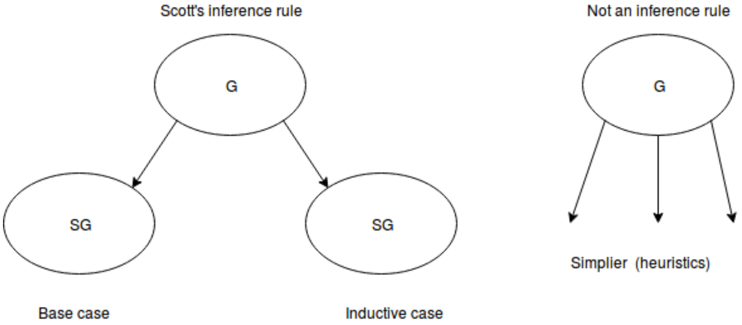


Figure : Some commands for backwards proof in LCF

Backward proof

- Goal seeking activities appear in different fields. For example: Artificial Intelligence.
- Milner worked at the time for the Stanford Artificial Intelligence Laboratory.
- In mathematics it relates with the method of backward proof. Let's see an example:

Theorem (*Geometric and arithmetic means inequality*)

If $x, y \in \mathbb{R}^+$ *and* $x \neq y$ *then* $\frac{x+y}{2} > \sqrt{xy}$.

Proof.

$$
\begin{aligned}
\frac{x+y}{2} > \sqrt{xy} &\equiv \frac{(x+y)^2}{4} > xy \equiv && \text{x,y positive} \\
&\equiv x^2 + 2xy + y^2 > 4xy \equiv && \text{by algebra} \\
&\equiv x^2 - 2xy + y^2 \equiv && \text{by algebra} \\
&\equiv (x-y)^2 > 0 \equiv && \text{by algebra} \\
&\equiv \mathit{true} && \text{x,y are different and} \\
& && \text{square is positive}
\end{aligned}
$$

$\square$

Edinburgh logic of computable functions

The paper addresses two major issues that appeared in Standford LCF:

- The lack of any way for users to add new proof commands.
- Large proofs could exhaust available memory.

His ideas were implemented in the Edinburgh LCF (went to Edinburgh in 1973).

1 The paper's history and its importance at the time

2 A theory of goal seeking

3 An example: a theorem about parsing

4 Posterior developments

5 Conclusion and references

Introducing new concepts in the design of proof assistants

The paper presents many of the core ideas underlying modern proof assistants:

- Functional programming language as a proof assistant metalanguage.
- Representing terms, formulae and theorems of a logic as metalanguage data types.
- Programming language types to ensure soundness.
- Tactics as metalanguage functions for representing subgoaling strategies.
- Higher-order functions for combining tactics.

Milner's theory of tactics

The theory of tactics relies on the following definitions:
Let's denote the type of formulas by F, the type of goals by G, the type of procedures by P and the type of events by E.

Name	Type	Notation	Meaning
goal	List[F] $\times$ F	$(\Gamma$,F)	$\Gamma \vdash F$ unproved
event	List[F] $\times$ F	$(\Delta$,G)	$\Delta \vdash G$ proved
tactic	G $\rightarrow$ List[G] $\times$ P	T(G) $=$ ([$G_1; \cdots ; G_n$],P)	
procedure	List[E] $\rightarrow$ E	P([$E_1; \cdots ; E_n$])	
achieve	E $\times$ G	achieves((Δ,G),(Γ,F))	$\exists \Gamma' \subseteq \Gamma$:($\Delta$,G) $=$ (Γ',F)

- Equality for achieves relation is up to rename of bound variables.
- Functions here are considered to be partial functions.

Milner's theory of tactics

Definition (Validity of tactics)

A tactic T is valid if whenever $T(G) = ([G_1; \cdots ; G_n], P)$ and if E_i achieves G_i then $P([E_1; \cdots ; E_n])$ is defined and achieves G.

Definition (Solving a goal)

If a valid tactic T verifies $T(G) = ([], P)$ then T is said to solve the goal G and $P([])$ should evaluate to an event that achieves G.

Using types to ensure logical soundness

What is the relation between a goal and events?

- The two have a type $List[F] \times F$
- However, an event has an abstract type (meaning it cannot be instantiated directly).
- ML type-checker ensures that only functions representing sound inference rules in Scott's logic can create new events.

Saving space by not storing proofs

- Standford LCF created a proof tree stored in computer's memory.
- Each proof command expands this tree by adding generated subgoals.
- The validation procedure solves exhaustion of memory problems.

Milner's theory of tactics

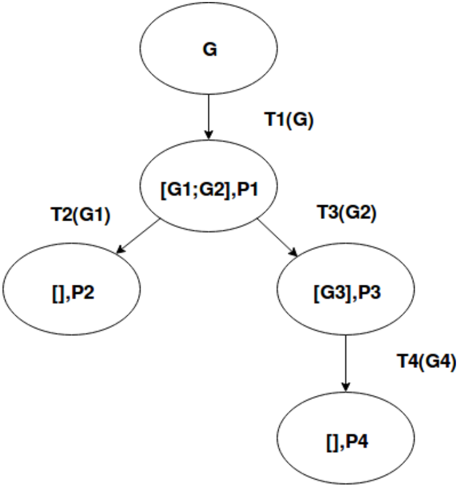


Figure : Tactics applications

Milner's theory of tactics

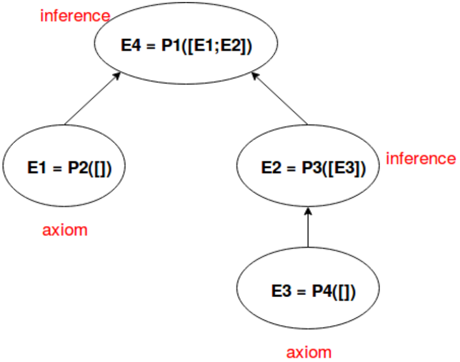


Figure : Validations applications

Examples of tactics

Name	Goal	Subgoal	Validation
GENTAC	$(\Gamma, \forall x.F)$	(Γ, F)	$x \notin free(\Gamma)$
DISCHTAC	$(\Gamma, A \implies B)$	$(\Gamma \cup A, B)$	implication rule

Table : Primitive tactics

Tacticals

- Allow to introduce new proof commands in LCF (solves first issue).
- Requires higher-order functions in metalanguage ML.
- Combination of tactics require exception-handling mechanism for inapplicable tactics.
- Example: induction tactic applied to goal that cannot be decomposed into basis and step subgoals.

Example of tacticals

Let's denote by T the type of tactics.

Name	Argument type	Meaning	Validation
THEN	$T \times T$	flatten$(T_2(T_1(G)))$	$P_1 \circ P_2$
THENL	$T \times List[T]$	flatten$(List[T]$ map $T_1(G))$	$P_1 \circ P_i$
ORELSE	$T \times T$	$(T_1.getOrElse(T_2))(G)$	succeed(P)
REPEAT	T	map T until failure or $\emptyset$	$P^{\text{successes}}$

Table : Simple tacticals

Composition of tacticals

- A tactic that repeatedly strips off leading universal quantifiers and assumes the antecedent of any implication in the goal formula.
- A goal $(\Gamma, \forall x.F_1 \implies \forall y, z.F_2 \implies F_3)$
- Transformed into $(\Gamma \cup \{F_1, F_2\}, F_3)$.
- REPEAT (GENTAC ORELSE DISCHTAC)

Complex tactics and tacticals

- Some complex tactics and tacticals that were not presented before:
    - Full automatic proof methods can be implemented as tactics.
    - Example: RESTAC (Robinson resolution method)
    - SIMPTAC: a simplification tactic based upon a collection of equational theorems of the form $\Gamma \vdash t_1 = t_2$.
    - Tactics implementing induction and case analysis.
- Specialized tactics leading to a "tower" of theories (discussed later).
- Special importance may have polymorphic theories. Example: trees with parametrized nodes.

Using types to ensure logical soundness

- The validation procedures produced by tactics actually create new theorems.
- How are we sure that the produced theorems are correct?
- The key again is that the type system does not allow to generate a false event.

1 The paper's history and its importance at the time

2 A theory of goal seeking

3 An example: a theorem about parsing

4 Posterior developments

5 Conclusion and references

Notation

- We introduce a slightly more complex notation for sequents (Γ,F) as (Γ,F,S)
- S is a set of equational assumptions for simplification by SIMPTAC
- Other rules may add suitable equations to S
- Example: if DISCHTAC has an equational antecedent then it is added to Γ and S.

The tower of theories

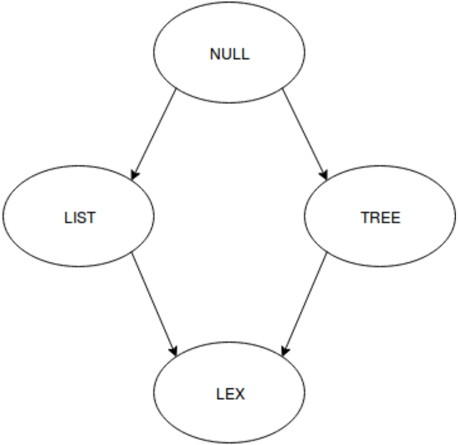


Figure : The tower of theories in our example

The null theory

We create the different theories as daughters of the NULL or pure theory. This theory:

- Does not contain non-logical types, constants or axioms.
- Possesses a type ONE.
- The only proper member object is denoted by the constant "nothing" symbol ().

Polymorphic theory of lists

- Define a unary type constructor, LIST.
- Introduce two constants with polymorphic types:
  - NIL: α LIST
  - CONS: $\alpha \rightarrow \alpha$ LIST $\rightarrow \alpha$ LIST
- Define function APPEND: α LIST $\rightarrow \alpha$ LIST $\rightarrow \alpha$ LIST.
- We characterize the LIST constructor by the isomorphism: α LIST $\simeq$ ONE $+ \alpha \times \alpha$ LIST.
- We characterize the constants writing equations such as: APPEND(CONS a x) y = CONS a (APPEND x y)

Polymorphic theory of trees

- Define a binary type constructor, TREE.
- We characterize the TREE constructor by the isomorphism: (α, β) TREE $\simeq \alpha + \beta \times$ (α, β) TREE $+ \beta \times$ (α, β) TREE $\times$ (α, β) TREE.
- Introduce three constants for the three tree constructors:
  - TIP: $\alpha \to (\alpha, \beta)$ TREE
  - NODE1: $\beta \to (\alpha, \beta)$ TREE $\to (\alpha, \beta)$ TREE
  - NODE2: $\beta \to (\alpha, \beta)$ TREE $\to (\alpha, \beta)$ TREE $\to (\alpha, \beta)$ TREE
- Computation induction rule of Scott's logic provide us automatically with an induction rule for lists and trees. We will use a TREEINDUCTTAC in this example.

Lexical theory

- Define types ID and OP for variables and operators in the list to be parsed.

- Define a type isomorphism that takes account of symbols (and) and the occurrence of an operator as a unary or binary function:
  $\mathrm{SYMB} \simeq \mathrm{ONE} + \mathrm{ONE} + \mathrm{ID} + \mathrm{OP} + \mathrm{OP}$

- We introduce and axiomatize five constructors:
  - LB :SYMB
  - RB :SYMB
  - VAR :ID $\rightarrow$ SYMB
  - UNARY :OP $\rightarrow$ SYMB
  - BINARY :OP $\rightarrow$ SYMB

Parsing algorithm

- We express the parsing algorithm as a set of axioms about a function PARSE.
- The type of PARSE is PARSE: SYMB LIST $\rightarrow$ (ID,OP) TREE $\times$ SYMB LIST.
- The symbol list begins with a well formed formula.
- The resulting pair is made with the shortest initial segment of the argument that is parsable, together with the remainder of the argument string.
- A parse tree will be a tree with tips labelled in ID and whose nodes are labelled in OP.

Parsing algorithm

- We express the parsing algorithm as a set of axioms about a function PARSE.
- The type of PARSE is PARSE: SYMB LIST $\to$ (ID,OP) TREE $\times$ SYMB LIST.
- The symbol list begins with a well formed formula.
- The resulting pair is made with the shortest initial segment of the argument that is parsable, together with the remainder of the argument string.
- A parse tree will be a tree with tips labelled in ID and whose nodes are labelled in OP.

Parse example

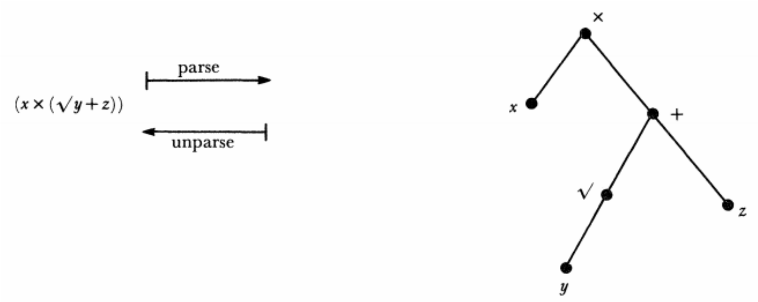


Figure : An example of parse and unparse

Axioms for PARSE

Take $x \in ID$, $f \in OP$, $s \in SYMBLIST$ and $t \in (ID, OP)TREE$.
Then:

- PARSE (CONS (VAR x) s) = (TIP x, s)

- PARSE (CONS (UNARY f) s) = (NODE1 f t',s')
  WHERE (t',s') = PARSE s

- PARSE (CONS LB s) = PARSETWO t' s'
  WHERE (t',s') = PARSE s

- PARSETWO t (CONS (BINARY f) s) =
  (NODE2 f t t', CHECKRB s')
  WHERE (t',s') = PARSE s

- CHECKRB (CONS RB s) = s

Axioms for UNPARSE

- The type of UNPARSE is UNPARSE:(ID,OP) TREE $\rightarrow$ SYMB LIST.

- UNPARSE (TIP x) = CONS (VAR x) NIL

- UNPARSE (NODE1 f t) =
  CONS (UNARY f) (UNPARSE t)

- UNPARSE (NODE2 f t t') =
  APPEND (CONS LB (UNPARSE t))
  (APPEND (CONS (BINARY f) (UNPARSE t'))
  (CONS RB NIL)))

The theorem

We want to proof the following theorem over a parsing problem:

F:$\forall t, s.$ WD[t] $\implies$ PARSE (APPEND (UNPARSE t) s) = (t,s)

Where WD[t] expresses that the tree t is appropriately well defined.

The tactic employed

The intuition needed from the user is that is natural to attack this problem by:

- Structural induction upon trees.
- Mixture of simplification by SIMPTAC and routine logical manipulation by RESTAC.
- As implication and quantification are used: GENTAC and DISCHTAC.
- It is reasonable to expect that WD[t] has been proved in advanced in OTHER THEORIES.

Let's represent by L the set of auxiliary lemmas coming from WD[t].

The tactic goes as follows:

```
USELEMMASTAC (L) THEN
TREEINDUCTAC THEN SIMPTAC THEN
REPEAT (GENTAC ORELSE DISCHTAC) THEN
RESTAC THEN SIMPTAC
```

It is claimed that:

> *The combination of tactics used was remarkably similar to that which succeeded on other problems, in totally different problem domains.* **The difference was mainly in the particular induction rule used, in the particular auxiliary lemmas employed and in the particular simplification rules embodied in the main goal G.**

1 The paper's history and its importance at the time

2 A theory of goal seeking

3 An example: a theorem about parsing

4 Posterior developments

5 Conclusion and references

Influences in other proof assistants

LCF influenced the creation of posterior proof assistants such as:

- Cambridge LCF (Paulson, 1987): enriched LCF logic and rewriting and tactics techniques.
- Nuprl, Constructions and Petersson's Programming system for Type Theory
  - Based on the LISP implementation of ML in Edinburgh LCF.
  - Tactics were modified to validate goals with representations of proofs rather than theorems.

Influences in other proof assistants

LCF influenced the creation of posterior proof assistants such as:

- HOL (Gordon, 1993): and
  - Based on the LISP code implementing Cambridge LCF.
  - Tactics were adapted for classical higher-order logic rather than for Scott's logic.
- Isabelle (Paulson, 1990):
  - A generic tool to create proof assistants by instantiating a logical framework.
  - No need to create a system for each modification of LCF.
  - Tactics and tacticals operate on proof states rather than on subgoals.

How tactics survived LCF

On the one hand...

- Scott's logic was less suitable for verifying hardware or checking general mathematical proofs.
- It was tactics and typed programmable metalanguage for ensuring logical soundness rather than LCF logic that had a major impact on modern theorem proving.
- Descendants of LCF underlie the majority of proof assistants today.

on the other hand...

- Programming algorithms as derived rules or tactics is more complicated than programming them directly.
- Resulting implementations had poor performance.
- PVS or ACL2 showed better performance than LCF systems.
- Does LCF's performance penalty worth logical soundness?

Progress in machine checked proofs

Some machine proofs made with tactic-based proof assistants:

- HOL: ARM6 processor (2003)
- Coq: four colour theorem (2007), cryptographic proofs (Barthe et alii., 2009), CompCertC compiler (2009), Feit-Thomson theorem (2013)
- Isabelle: correctness of security protocols (Paulson, 1998), seL4 operating system(2009), Gödel's second incompleteness theorem (2014)
- HOL/Isabelle: Kepler conjecture on sphere packing (2014)

The tower of theories

In the paper, Milner establishes the need to keep a "tower of theories" that would...

- Allow us to define different knowledge in each problem or theory where we are working.
- Allow us access to those theorems previously defined.
- The use of this "tower" should decrease the time needed to prove new results.

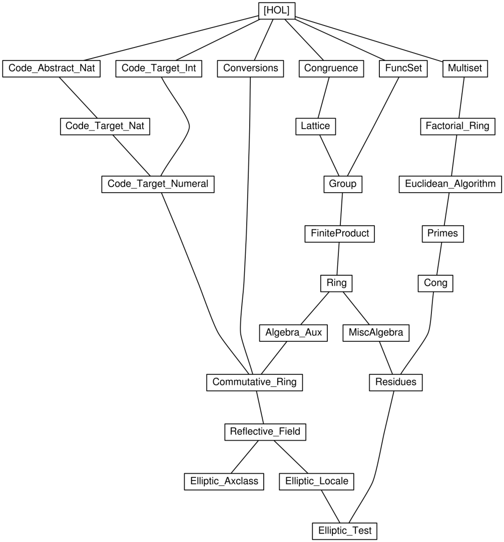


Figure : Theory tower for a proof about elliptic curves in Isabelle

Inconveniences of the tower of theories

The model of the tower of theories has of course limitations:

- Every new proof assistant should start its theories from scratch...
- unless we accept the use of different proof assistants for proofs.
- The proof that two different proof assistants can be combined to obtain certain results is not necessarily trivial.

Influence in programming languages

The characteristics that we have explored on ML made it suitable as a base for other functional languages such as:

- Standard ML (metalanguage in Isabelle and several HOL systems)
- OCaml (metalanguage for Coq and HOL Light)
- F# initially derived from OCaml

1 The paper's history and its importance at the time

2 A theory of goal seeking

3 An example: a theorem about parsing

4 Posterior developments

5 Conclusion and references

Conclusion

"It emerges from these various studies that the method of composing proof tactics, which is illustrated in this paper on a rather simple example, not only provides a means of communicating proof methods to a machine, and of tuning them to particular needs, but also presents to mathematicians and engineers **a lucid way of communicating such methods among themselves***."*

Intuition $\rightarrow$ Formalization?

References

M.J.C. Gordon. "Tactics for mechanized reasoning: a commentary on Milner (1984)'The use of machines to assist in rigorous proof'." In: *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* (2015), p. 20140234.

Robin Milner. "The use of machines to assist in rigorous proof [and discussion]." In: *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* (1984), pp. 411–422.

L.C. Paulson. *Logic and computation: interactive proof with Cambridge LCF*. Cambridge University Press, 1990.

Questions?