On Extensibility of Proof Checkers *

Robert Pollack

Dept. of Computing Science
Chalmers Univ. of Technology and Univ. of Göteborg
S-412 96 Göteborg SWEDEN
pollack@cs.chalmers.se

1 Introduction

This paper is about mechanical checking of formal mathematics. Given some formal system, we want to construct derivations in that system, or check the correctness of putative derivations; our job is not to ascertain *truth* (that is the job of the designer of our formal system), but only *proof*. However, we are quite rigid about this: only a derivation in our given formal system will do; nothing else counts as evidence! Thus it is not a collection of judgements (provability), or a consequence relation [Avr91] (derivability) we are interested in, but the derivations themselves; the formal system used to *present* a logic is important. This viewpoint seems forced on us by our intention to *actually do* formal mathematics.

There is still a question, however, revolving around whether we insist on objects that are immediately recognisable as proofs (*direct* proofs), or will accept some meta-notations that only compute to proofs (*indirect* proofs). For example, we informally refer to previously proved results, lemmas and theorems, without actually inserting the texts of their proofs in our argument. Such an argument *could* be made into a direct proof by replacing all references to previous results by their direct proofs, so it might be accepted as a kind of indirect proof. In fact, even for very simple formal systems, such an indirect proof may compute to a very much bigger direct proof, and if we will only accept a fully expanded direct proof (in a mechanical proof checker for example), we will not be able to do much mathematics. It is well known that this notion of referring to previous results can be internalized in a logic as a *cut* rule, or Modus Ponens. In a logic containing a cut rule, proofs containing cuts are considered direct proofs, and can be directly accepted by a proof checker[2].

Another example is the deduction theorem. In the Hilbert style presentation of minimal implicational logic, with constructors S, K, and Modus Ponens, if proposition a can be derived from a set of assumptions, G, (notation $G \vdash a$), then a derivation of $b \rightarrow a$ from G with b removed, $G \backslash b \vdash b \rightarrow a$, can be constructed. The former derivation can be much shorter [Kle67], and might be accepted as an indirect notation for the latter. As in the case of the cut rule,

* This work was supported by the ESPRIT BRA on TYPES and by the British SERC, and was partly done at the University of Edinburgh

[2] This example should be compared with the notion of *definition* as internalized in Automath [dB85].

this indirectness can be removed by internalizing the deduction theorem as the implication introduction rule of a natural deduction presentation of the logic. In this case, however, the internalization requires either a new notion of formal system (natural deduction with discharge) or a new language for judgements (sequents instead of formulas).

As a final introductory example, consider a natural deduction or sequent style presentation of classical first order logic. It is well known that every propositional tautology is a theorem in this logic, and that a direct proof of a tautology can be mechanically constructed. Thus "by tautology" is often used as an indirect proof notation for classical first order logic, and this can be internalized as direct proof by adding a new axiom scheme saying that P is an axiom under the side condition that P is a tautology. In the two previous examples of internalizing indirect proof notations, it was computationally trivial to test whether the internalized, direct proof rules, were applicable (although the computation to expand out the new rules is not trivial); in this case the side condition which must be checked, "P is a tautology", is non-trivial. This axiom scheme is usually called a *decision procedure*, and I suggest that all decision procedures can be viewed as axioms or rules with computationally non-trivial side conditions.

We see that choosing a formal system (presentation of a logic) with an expressive notion of direct proof is very important if we hope to actually check interesting examples in reasonable time and space. However, if we are to do formal mathematics, we must choose some formal system in which to work[3], and experience shows it is unlikely we can capture directly all the proof notations that will be convenient for the work of formalization. Furthermore, there are other criteria for our proof checker, such as simplicity and correctness of the implementation, that mitigate for a syntactically simple formal system, with few direct proof notations. Thus we have the problem that this paper is about: how to construct a mechanical proof checker such that the stringent requirement of formality can be met without sacrificing the convenience and computational efficiency of such indirect proof notations as new rules and decision procedures.

The approach I suggest is to represent the formal system to be implemented, the *object* system, within a meta system having certain computational and logical expressiveness, and to accept proof notations that are *non-canonical* (not computed to normal form) in the meta system, but that we can prove in the meta system will compute to (representations of) direct proof notations for the object system. That is, the imprecise notion of "indirect proof notation" discussed above, is made precise as the notion "non-canonical" in a formal meta system.

What is all this obscure discussion about? Haven't I just proposed the LCF [GMW79] notion of *tactic*, a program in some computational meta language that is proved (by properties of the meta language type system) to compute a direct object language proof, and which is used to allow extensibility of a proof checker without compromising the simplicity and safety of the implementation?

In order to clarify what is new in my proposal I will present a running ex-

[3] I do not suggest that any single system will do for all of mathematics.

ample, implementing a very simple formal system in three different ways, and discuss which proof notations for the example system can be represented in each of the three implementations, and the computational cost of using such proof notations. After a digression on the nature of *rules* in formal systems (section 1.1), I present the formal system of my running example (section 1.2), give a pure LCF implementation of this system (section 2), an implementation in an LCF variant style inspired by Nuprl [Con86] (section 3), and an implementation using a meta language with a much more expressive logic than the type system of ML, the LCF meta language (section 4). I conclude by discussing a few pragmatic aspects of actually using my proposal, and related work.

1.1 Rules of Inference

In particular formal meta systems the notions of "formal system" and "rule of inference" are concrete, but in order to compare meta systems I give an abstract presentation. My reference for this subsection is [HS86].

Let $\mathcal{L}$ be the language of judgements[4] of some formal system. A *rule of inference* is a partial function from lists over $\mathcal{L}$ into $\mathcal{L}$. If $\mathcal{R}$ is a rule, $A_1, \ldots, A_n, B \in \mathcal{L}$, and $\mathcal{R}[A_1, \ldots, A_n] = B$, we say

$$\frac{A_1, \ldots, A_n}{B}$$

is an *instance* of $\mathcal{R}$, with *premises* $A_1, \ldots, A_n$ and *conclusion* B. An *axiom* is a rule all of whose instances have an empty list of premises.

Remark. Some presentations of this material generalize the notion "rule" to be a decidable relation between lists over $\mathcal{L}$ and $\mathcal{L}$. This generalized notion better captures the informal use of rules; i.e. side conditions may refer to the conclusion as well as to the premises. The choice is largely a matter of taste, as one relational rule can be replaced by a (possibly infinite) number of functional rules. In fact, with functional rules, an axiom has only one instance, whereas a rule may have many instances, and this artefact of the abstraction is inelegant. I have chosen the functional notion of rule because that is the notion that best explains LCF.

Also, some presentations of this material, including [HS86], restrict each rule to a fixed number of premises. Whether this matters at all depends on the meta language used to express rules, e.g. Feferman [Fef88] shows how to reduce the notion "rule with a list of premises" to "rule with two premises" in a particular formal meta system. My choice to allow a list of premises (of unspecified length) is due to our interest in actual machine implementations. For example, HOL [Gor88] has a rule for simultaneous substitution having a list of premises.

Let $\mathcal{F}$ be a formal system whose language of judgements is $\mathcal{L}$. A rule, $\mathcal{R}$, is *derivable* in $\mathcal{F}$ if, for every instance of $\mathcal{R}$, there is a deduction in $\mathcal{F}$ of its

[4] See [Mar85] about the notion of *judgement*. I have in mind sequents, as well as propositions, for judgement forms.

conclusion from its premises.

$$\frac{\begin{array}{c}\text{premises of } \mathcal{R} \\ \vdots \\ \vdots \\ \vdots\end{array}}{\text{conclusion of } \mathcal{R}}$$

Clearly adding derivable rules to a formal system does not change its set of theorems (judgements provable from no assumptions); and if $\mathcal{R}$ is derivable in $\mathcal{F}$, then $\mathcal{R}$ is derivable in any extension of $\mathcal{F}$ by new axioms and rules.

A rule, $\mathcal{R}$, is *admissible* in $\mathcal{F}$ if, for every instance of $\mathcal{R}$, whenever its premises are provable (from no assumptions), then so is its conclusion. Clearly every derivable rule is admissible, and adding admissible rules to a formal system does not change its set of theorems. In fact, $\mathcal{R}$ is admissible in $\mathcal{F}$ iff adding $\mathcal{R}$ to $\mathcal{F}$ as a new rule does not change $\mathcal{F}$'s set of theorems. Thus it is exactly the set of admissible rules that we would like our proof checker to accept indirect notations for. Unlike derivability in $\mathcal{F}$, the notion of admissibility in $\mathcal{F}$ is not necessarily preserved by extending $\mathcal{F}$ with new axioms or rules. Finally notice that every admissible axiom is also derivable, although it is not true in general that every admissible rule is also derivable.

Surprisingly, although LCF allows arbitrary computation in tactics, it does not have tactics for admissible rules that are not derivable. Nuprl had already fixed this problem in the mid 1980's, but the tactics for the new rules are infeasible to actually execute, at least in worst cases. In my suggested approach, tactics which are proven correct do not have to be executed, although the side conditions which verify that a tactic is applicable in a particular case, such as the tautology checking decision procedure mentioned in the introductory examples, do have to be executed.

1.2 The example formal system

The formal system of my running example is a natural deduction presentation of minimal implicational logic:

$$\text{AXIOM}\quad \frac{}{G \vdash a} \qquad a \in G$$

$$\text{INTRO}\quad \frac{G, a \vdash b}{G \vdash a \rightarrow b}$$

$$\text{ELIM}\quad \frac{G \vdash a \rightarrow b \quad G \vdash a}{G \vdash b}$$

Notice that, by our definition, AXIOM is an axiom *scheme*, with one axiom for each pair a, G such that $a \in G$. (If we used the relational notion of rule, AXIOM could be a single axiom.) On the other hand INTRO and ELIM are both single rules. I will not be so careful about such matters in the rest of this paper.

2 LCF

LCF [GMW79] is a proof checker for a particular logic of computation developed during the 1970's. It was so original, presenting a whole concept of how to organize the implementation of a proof checker, that it is mainly cited for its contribution to the technology of mechanical proof tools. One of the most widely used proof systems today, HOL [Gor88] is a direct descendent of LCF, implemented with pure LCF-style; other proof systems, e.g. Nuprl [Con86], Isabelle [Pau93] and LAMBDA [FFM90], are variations on LCF; and almost all are influenced by LCF. The main features of an LCF-style implementation are:

- Its soundness as a proof checker depends on a relatively small, isolated section of code.
- It allows users to program new proof search techniques (*tactics*) using essentially arbitrary computation, without compromising soundness of the proof checker.
- It supports *refinement* proof, i.e. top down proof, refining a goal to subgoals, and refining the subgoals in turn.

Today, the use of *existential variables* [Mil92] and unification in proof systems makes refinement proof less problematic, and the LCF technique for refinement can be simplified (e.g. the LCF variants Isabelle and LAMBDA). However, when it comes to extensibility, allowing new proof search techniques to be programmed without compromising soundness of the proof checker, most ideas being explored today are variations on LCF-style.

2.1 The Kernel and Foward Proof

Here is an LCF-style implementation of the example formal system (section 1.2) in Standard ML (SML) [MTH90]. The syntax of *propositions* is given by an inductive type, generated by *propositional variables* and *implication*; *contexts* are lists of propositions; the goals we wish to prove have the shape $G \vdash a$ where G is a context and a is a proposition.

```
datatype prop = pv of string                (* propositional vars ...      *)
              | arr of prop*prop;           (* ... implication             *)
type cxt = prop list;                       (* contexts of assumptions     *)
type goal = cxt*prop;                       (* shape of judgements  G |- a *)
```

I use one exception, `Failure` to signal all errors.

```
exception Failure of string;
fun fail s = raise Failure s;
```

Now the kernel of the proofchecker.

```
abstype thm = True of goal                                    (* hidden  *)
with fun destr_thm (True j) = j                               (* visible *)
     fun Axiom G a = if member a G then True(G,a) else fail"Axiom"
     fun Intro J = case destr_thm J
                    of (a::G,b) => True(G,arr(a,b))
                     | _        => fail"Intro: malformed premiss"
     fun Elim J K = case (destr_thm J,destr_thm K)
                     of ((G,arr(a,b)),(H,c)) =>
                                  if G=H andalso a=c then True(G,b)
                                  else fail"Elim: malformed premiss"
                      | _ => fail"Elim: malformed premiss"
end;
```

The type **thm** is abstract: its constructor, **True**, is not visible outside the **abstype**, so the only way for a user of this abstype to create a **thm** is to use one of the three functions, **Axiom:cxt->prop->thm**, **Intro:thm->thm**, and **Elim:thm->thm->thm**, that are exported by the **abstype**. This kernel also supplies a function **destr_thm** for users to access the goal proved by a **thm**.

The functions **Axiom**, **Intro**, and **Elim** construct proofs in a forward direction, from the leaves to the root. For example, given a **cxt**, G, and a **prop**, a, **(Axiom G a)** returns **True(G,a)** if this is justified, and raises **Failure"Axiom"** otherwise.

Remark. To make clear that **Intro** and **Elim** are inference rules and **Axiom** is an axiom scheme (section 1.2), we could package them slightly differently:

```
type proofRule = (thm list)->thm
fun Axiom G a [] =  ....     ;   Axiom : cxt->prop->proofRule;
fun Intro [J] =  ....        ;   Intro :            proofRule;
fun Elim [J,K] =  ....       ;   Elim  :            proofRule;
```

These rules can only be used to build proofs (from no assumptions), not deductions.

These basic forward proof constructors are supposed to be correct implementations of the three rules of the logic, and it is usually implied that if this small section of code is correct, then the implementation is perfectly safe: we can allow unrestricted use of **Axiom**, **Intro**, and **Elim** without endangering soundness of the datatype **thm** with respect to the logic. Notice however that **Axiom** uses the **member** function, and if, for example, **member** always returned **true**, our LCF kernel would implement a very different logic than expected. Unless the **abstype** is closed, correctness may depend on other code as well.

It is clear that this implementation is complete for the logic: every proof that we believe is correct can be built using the functions **Axiom**, **Intro**, and **Elim**. However, when we speak of expressiveness for tactics below, we will find it is not completeness for theorems that we want, but completeness for *admissible rules*.

2.2 Tactics and Refinement Proof

Another important idea of LCF is a simple technique to implement backwards, refinement proof, from the forward proof constructors. A *tactic* is a function that given a goal, returns a list of new goals (subgoals) and a function called a *validation*.

```
type validation = thm list->thm;
type tactic = goal -> (goal list*validation);
```

The idea is that the validation produces a (forward) proof of the original goal, given (forward) proofs of the subgoals. Thus `validation` is the type of inference rules (section 1.1).

First we have basic tactics, one for each of the forward proof constructors:

```
fun axiom_tac (G,b) = ([],
                       fn [] => Axiom G b | _ => fail"axiom_tac");
fun intro_tac (G,arr(a,b)) = ([(a::G,b)],
                              fn [th] => Intro th | _ => fail"intro_tac")
  | intro_tac _ = fail"intro_tac";
fun elim_tac a (G,b) = ([(G,arr(a,b)),(G,a)],
                        fn [thl,thr] => Elim thl thr
                         | _         => fail"elim_tac");
```

For example, `intro_tac`, given a goal $G \vdash a \rightarrow b$ returns a single subgoal $G, a \vdash b$, and a validation that, when eventually passed an object, `th` of type `thm` (packaged in a singleton list), returns `(Intro th)`. If `th` is really a proof of $G, a \vdash b$, then `(Intro th)`, is really a proof of $G \vdash a \rightarrow b$. The validation will be executed (applied to a `thm list`) at a later time than the tactic itself is executed.

Remark (Early failure of refinement steps.). Note that `(axiom_tac (G,b))`, which always succeeds, may return a validation that will certainly fail, e.g. when $b \notin G$. While the implementation above is correct, we will keep the users happier if we do as much checking as possible when the tactics are executed, and not wait for the validations to execute. Thus a better version of `axiom_tac` is given by:

```
fun axiom_tac (G,b) = if not (member b G) then fail"axiom_tac"
                      else ([], fn [] => Axiom G b
                                 | _  => fail"axiom_tac");
```

With this version of `axiom_tac`, `(member b G)` is checked twice, first when the tactic is executed, and later when the validation (which calls `(Axiom G b)`) is executed. In LCF variants such as Isabelle and LAMBDA, this duplicate test is not needed.

Remark (Deferred instantiation of rule premises.). The rule ELIM does not have the subformula property because a does not appear in the conclusion, so `elim_tac` must be passed this information explicitly. It is an artifact of the details of LCF that one cannot defer such choices until later. The LCF variant Isabelle has

existential variables in its object language[5], allowing deferred instantiation in cases like `elim_tac`. However, we must ask whether the safety of such an implementation depends on correctness of the algorithm it uses to instantiate these existential variables, which is higher-order unification in Isabelle. The kernel of an LCF style implementation is supposed to be simple, its correctness self evident, which is not the case for higher order unification programs.

Proof tools having existential variables in their language, such as ALF, Coq and LEGO, also allow deferred instantiation of rule premises. For the three tools cited, correctness does not depend (at least in principle) on correctness of a unification algorithm, as they produce fully instantiated proof objects that can be checked without using any tactics, search, or unification algorithms.

2.2.1 What can go wrong with tactics? A tactic (to be more precise, a validation) can never construct a non-theorem, but two things can go wrong:

It might return the wrong theorem. The type system is not expressive enough to say which theorem a tactic returns. For example

```
fun wrong_thm_tac (G,b) = ([], fn [] => Axiom [b] b);
```

has type `tactic`, and when executed returns the validation

```
fn [] => Axiom [b] b
```

This validation, when executed, returns the theorem $b \vdash b$. By convention, we expected the tactic to refine the goal it was called with, `(G,b)`, but it doesn't actually do so. We must execute a validation to completion and examine the `thm` object it returns to be sure it is the intended one.

It might not return at all. A function of type `validation`, i.e. type `(thm list->thm)`, can fail to return a `thm` object in at least two ways. Firstly, using general recursion, it can start some potentially infinite computation; e.g. a tactic that begins enumerating all derivations searching for a proof of the current goal. Secondly, it can raise an exception; in fact the basic tactics, `axiom_tac`, `intro_tac` and `elim_tac`, must call the atomic forward proof constructors, `Axiom`, `Intro` and `Elim`, which themselves can raise exceptions. The atomic forward proof constructors must be able to raise exceptions, as the ML type system is not expressive enough to limit their application to correct instances. For example, there is no way to express the side condition of rule AXIOM, that $a \in G$, in the typing of `Axiom`[6].

[5] Some confusion may arise as the Isabelle object language is intended to be used as a meta language for representing other languages and logics in a *framework* style.

[6] We could program LCF in a language of total functions without exceptions, say F^ω. In F^ω, lacking dependent types, the atomic forward proof constructors must be able to signal failure, for the reason mentioned above, but there are other ways to signal failure besides exceptions (see section 5.3). These possibilities don't change my basic argument; validations would still have to be evaluated to find out if they signal failure. The fundamental difficulty is that in F^ω, as in ML, the type system cannot specify the forward proof constructors precisely enough to avoid the need to fail. In section 4 we will see an alternative that solves this fundamental difficulty.

For these reasons, tactics, and the validations they return, must be evaluated in order to be sure they actually terminate and build a proof of the intended goal.

2.3 Tacticals

There are functions, called *tacticals*, for combining tactics. They do the task of passing around the lists of new goals and the validations, and composing validations in the correct way. I will use the tacticals:

`(ORELSE tac1 tac2 g)` applies `tac1` to goal `g`, and if that fails (returns exception `Failure`), applies `tac2` to goal `g`.

`(THEN tac1 tac2 g)` applies `tac1` to goal `g`, then maps `tac2` to the list of subgoals produced by `tac1`.

`(THENL tac1 tacs g)` applies `tac1` to goal `g`, then applies the list of tactics, `tacs`, elementwise to the list of subgoals produced by `tac1`.

`(ID_tac g)` refines any goal, `g` into one subgoal, namely itself. `ID_tac` is really a tactic, not a tactical, but its only use is in combination with tacticals such as `THENL`, where the list of tactics, `tacs`, might need to be padded out with dummy tactics.

2.4 Derivable Rules

Evidently LCF has tactics for all derivable rules. Consider the rule that, read as a refinement rule, forgets a hypothesis:

$$\text{FORGET}\qquad \frac{G \vdash b}{G \vdash a \rightarrow b}$$

We can see that this rule is derivable in our formal system by producing a derivation of its conclusion from its premise:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{G, b, a \vdash b}\,\text{AXIOM}}{G, b \vdash a \rightarrow b}\,\text{INTRO}}{G \vdash b \rightarrow (a \rightarrow b)}\,\text{INTRO} \qquad G \vdash b}{G \vdash a \rightarrow b}\,\text{ELIM} \tag{1}$$

The definition of derivable rule does not require us to build a *uniform*, or *schematic* deduction of the conclusion from the premises, as we did in this example; we may examine the premises, and then decide how to derive the conclusion. However, we are not given any derivation of the premises, so the deduction we build cannot depend on the shape of such a thing.

Here is a tactic for this rule.

```
fun forget_tac (G,arr(a,b)) =
    ([(G,b)], fn [th] => Elim (Intro (Intro (Axiom (a::b::G) b))) th
              | _     => fail"forget_tac")
  | forget_tac _ = fail"forget_tac: malshaped goal";
```

The interesting thing is the returned validation, which represents the deduction shown in (1) as the `thm` object

```
(Elim (Intro (Intro (Axiom (a::b::G) b))) th)
```

showing the tree structure of (1). More precisely, `thm`-objects are *not* derivations (deductions) from hypotheses, they are *proofs* from no assumptions. Thus, `forget_tac` (and the validation it returns) produce, not the derivation (1), but the proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{G,b,a\vdash b}\,\text{AXIOM}}{G,b\vdash a{\to}b}\,\text{INTRO}}{G\vdash b{\to}(a{\to}b)}\,\text{INTRO} \qquad \begin{array}{c}\vdots\\ J\\ \vdots\end{array} \qquad G\vdash b}{G\vdash a{\to}b}\,\text{ELIM}$$

However, `thm`-objects don't explicitly represent the structure of proofs, so a tactic or validation really cannot see inside the subproof J in this proof. Validations can only "glue proofs together", they cannot look at the structure of proofs; this is just enough to represent all derivable rules.

Remark. Tactics are often written using only the basic tactics, and tacticals. `forget_tac` would then look like:

```
fun forget_tac (g as (G,arr(a,b))) =
     (THENL (elim_tac b)
            [THEN intro_tac (THEN intro_tac axiom_tac), ID_tac]) g
  | forget_tac _ = fail"forget_tac: malshaped goal";
```

This version is independent of the details of implementation of refinement proof.

2.5 Admissible Rules

Is there a tactic for the rule

$$\text{WEAKENING}\quad \frac{K\vdash a}{G\vdash a} \qquad K\subseteq G$$

where the side condition $K\subseteq G$ means every member of K is also a member of G[7]? It is not obvious how to justify this rule by gluing proofs together; it

[7] In our notion of rule as a partial function (section 1.1), WEAKENING is not a rule, since G doesn't appear in the premises, but the rule schema "for every G there is a rule $K\vdash a\ /\ G\vdash a$ whenever $K\subseteq G$".

is justified by induction on the derivation of $K \vdash a$. Computationally we construct a new derivation following the shape of the given derivation of $K \vdash a$. For example

$$\dfrac{\dfrac{\dfrac{}{K, b, a \vdash b}\,\text{AXIOM}}{K, b \vdash a \to b}\,\text{INTRO}}{K \vdash b \to (a \to b)}\,\text{INTRO} \quad \Longrightarrow \quad \dfrac{\dfrac{\dfrac{}{G, b, a \vdash b}\,\text{AXIOM}}{G, b \vdash a \to b}\,\text{INTRO}}{G \vdash b \to (a \to b)}\,\text{INTRO}$$

We can see this always works because the AXIOM rule only looks for an occurrence of some proposition in the context, and if $a \in K \subseteq G$ then $a \in G$; no other rule depends on the context at all. More precisely, by induction on the structure of the derivation of of $K \vdash a$, with the base case handled by the reasoning above, and the induction cases trivial, we show that a program that stupidly traverses a proof of $K \vdash a$, replacing every K with G, will produce a correct proof of $G \vdash a$. Such a program doesn't need to know anything about our inductive proof of its correctness, but it does need to have access to the structure of the given derivation of $K \vdash a$. In the LCF-style implementation above, there is no tactic representing this justification of this rule, as the structure of the given derivation of $K \vdash a$ is not represented in any way. (Of course the trivial logic of our example is decidable, so we can just decide $G \vdash a$ without looking at $K \vdash a$ or its proof.)

Decision procedures We have just proved informally that $K \subseteq G$ is a decision procedure for the rule

$$\frac{K \vdash a}{G \vdash a} \tag{2}$$

i.e. that if we verify $K \subseteq G$ by computation, then we can safely replace a goal $G \vdash a$ by a goal $K \vdash a$ without any further checking. This proof of $G \vdash a$ is external to the system, and LCF, by design, does not let us construct a `thm` object from such external knowledge. Neither the tactic that actually builds a proof of $G \vdash a$ from a proof of $K \vdash a$, nor the decision procedure that merely replaces a goal $G \vdash a$ by a goal $K \vdash a$ are programmable in our LCF implementation.

Remark. I have argued that rules that are admissible but not derivable are not represented by any LCF tactic. This may not be as problematic as it sounds because derivability and admissibility coincide for axioms. For example, there is a tactic for the BDD decision procedure, which is the derivable axiom

$$\frac{\mathsf{BDD}(a) = \mathsf{BDD}(b)}{\vdash a = b}\,.$$

This tactic may use any representation of BDD, and any computation to test (informally prove) the side condition $\mathsf{BDD}(a) = \mathsf{BDD}(b)$. Then it returns a validation which is the computational part of an (informal) constructive proof that

$$\mathsf{BDD}(a) = \mathsf{BDD}(b) \quad \Rightarrow \quad \vdash a = b$$

instantiated with the informal proof that $\mathsf{BDD}(a) = \mathsf{BDD}(b)$; i.e. the validation must actually build the object proof of $\vdash a = b$.

Harrison [Har95a] describes a BDD decision procedure for HOL based on a different viewpoint. If I understand [Har95a] correctly, whereas I suggested making the test for $\mathsf{BDD}(a) = \mathsf{BDD}(b)$, and executing the proof that this test decides equality of expressions, both at the meta level, Harrison has proved $\vdash a = \mathsf{BDD}(a)$ as an object theorem in HOL, where $\mathsf{BDD}(\_)$ is some internal HOL representation. Little computation is necessary to use an object theorem, but serious computation is needed to eliminate the object level notion BDD. This seems close to *partial reflection* [How88] (which, in spite of the name, does not add any new principle to the object language), and the *metafunctions* of the Boyer-Moore prover [BM81], where some representation (BDD in this case) is shown in the object logic to be a correct "semantics" for some operation on some class of terms.

3 A Variation on LCF-style

We change the kernel of the LCF-style implementation above to make explicit the structure of derivations as part of the abstract datatype. This is the style used in Nuprl.

```
abstype thm = axiom of goal                                    (* hidden  *)
            | intro of goal*thm
            | elim  of goal*thm*thm
with                                                           (* visible *)
  datatype THM = AXIOM | INTRO of thm | ELIM of thm*thm
  fun destr_thm th = case th of axiom(g) => (g,AXIOM)
                              | intro(g,th) => (g,INTRO(th))
                              | elim(g,thl,thr)  => (g,ELIM(thl,thr))
  fun Axiom G a = if member a G then axiom(G,a) else fail"Axiom"
  fun Intro J = case destr_thm J
                 of ((a::G,b),_) => intro((G,arr(a,b)),J)
                  | _ => fail"Intro: malformed premiss"
  fun Elim J K = case (destr_thm J,destr_thm K)
                  of (((G,arr(a,b)),_),((H,c),_)) =>
                           if G=H andalso a=c then elim((G,b),J,K)
                           else fail"Elim: malformed premiss"
                   | _ => fail"Elim: malformed premiss"
end;
```

Now the datatype `thm` represents the inductive structure of derivations. But `thm` must be hidden, so we need another datatype, `THM`, in order for `destr_thm` to be able to return information about the structure of derivations.

The basic tactics and tacticals remain as before, and the tactic `forget_tac` works as before, but now it is possible to implement the weakening rule as a tactic because we have an explicit representation of derivations. First the definition of $\subseteq$:

```
fun forall f []     = true
  | forall f (h::t) = (f h) andalso (forall f t);
fun subcxt K G      = forall (fn b => member b G) K;
```

Now the tactic:

```
fun weaken_tac K (G,a) =
  if not (subcxt K G) then  fail"weaken_tac: not subcontext"
  else ([(K,a)],
        let fun weak L [th] =
          case destr_thm th
            of ((_,aa),AXIOM) => Axiom L aa
             | ((_,arr(b,_)),INTRO(th)) => Intro (weak (b::L) [th])
             | (_,ELIM(thl,thr)) => Elim (weak L [thl]) (weak L [thr])
        in weak G
        end);
```

The interesting part of this tactic is the validation it returns[8]. Just like `forget_tac`, this validation must be applied to one theorem. Unlike `forget_tac`, this validation actually builds a new tree by copying the shape of the one it is passed. In general, the new tree constructed by a tactic may be much larger than the trees it is applied to; e.g. the deduction theorem is exponential. As the development of some mathematical theory proceeds, the proofs that tactics must traverse get inexorably larger, and tactics such as the deduction theorem are surely not feasible to use in any but trivial examples.

Decision procedures In this LCF variant we have a tactic that first checks that $K \subseteq G$ and then returns a validation that builds a proof of $G \vdash a$ from a proof of $K \vdash a$. This validation is the executable part of the constructive meta proof (section 2.5) that

$$K \subseteq G \quad \Rightarrow \quad \frac{K \vdash a}{G \vdash a} \tag{3}$$

However the actual meta proof, our knowledge that the validation really implements (3), is external to the proof checker, and we still cannot code the decision procedure that checks $K \subseteq G$ then replaces a goal $G \vdash a$ by a goal $K \vdash a$ without building an object proof.

How expressive is it? By using a representation annotated with more information, including derivations rather than only judgements, we are able to express admissible rules as tactics that, by their very nature, we hope never to actually compute.

[8] The validation returned by `(weaken_tac K (G,a))` is supposed to compute the rule $K \vdash a \;/\; G \vdash a$ with the particular given `K` and `G`. However, to have strong enough recursion, we must define the rule scheme `weak` and then return the particular rule `(weak G)`. This shows again that the notion of rule as partial function is unsatisfactory.

4 LCF in a More Expressive Meta Language

It was mentioned above that LCF tactics may return the wrong theorem, or nothing at all, so we must actually run them to get their result. Now we will use a meta language with only total functions, so a well-typed tactic always terminates, and with dependent types, so the type of a tactic will say exactly what theorem (schema) it returns. The idea is that we won't have to run tactics at all, although determining their types will involve theorem proving. This idea is not due to me, and I briefly discuss related work below.

The meta language I will use for examples is Luo's Extended Calculus of Constructions (ECC) extended with inductive types [Luo94], as implemented in LEGO [LP92]. Other logics with inductively defined sets and total functions would do as well (see section 5.2).

I will show an implementation of the same formal system (section 1.2) as used in sections 2 and 3. The syntax of propositions and contexts is represented in LEGO much as in SML.

```
Inductive [prop:Prop]
    Constructors [pv:SS->prop]                (* propositional vars ...*)
                 [arr:prop->prop->prop];      (* ... implication       *)
[context = LL|prop];
```

The type of object language propositions, `prop`, is inductively defined, generated by propositional variables, `pv`, and implication, `arr`. The propositional variables range over a type, `SS`, with decidable (boolean valued) equality. In the SML examples I used type `string`, which is built-in to SML with boolean valued equality. In LEGO I am working parametrically in "any equality type". I will not discuss the machinary of equality types in LEGO (see [MP93, Pol94]). The type of contexts is lists (`LL`) of `prop`.

As in the second, more expressive LCF-variant, the theorems are inductively defined, but here they are an inductive relation instead of just an inductive type.

```
Inductive [thm:context->prop->Type]
  Constructors
[Axiom:{G|context}{a|prop}{sc:is_tt (member prop_eq a G)}thm G a]
[Intro:{G|context}{a,b|prop}  {J:thm (CONS a G) b}
                      (***********************************)
                                  thm G (arr a b)]
[Elim:{G|context}{a,b|prop}  {J:thm G (arr a b)}  {K:thm G a}
                      (***********************************)
                                  thm G b];
```

This definition says that `thm` has type `(context->prop->Type)`; if `G` is a `context` and `a` is a `prop`, then `(thm G a)` should be read as the type of proofs of the judgement $G \vdash a$. In this expressive language the types of `thm` objects say explicitly what theorem the object proves. Compare this with the ML type `thm` of the implementation in sections 2 and 3. The reader might object that in LCF one can compute the judgement proved by any given object of type `thm`, but this is

true only once the object is given, whereas the LEGO type `(thm G a)` is specific about which objects will be accepted as its members. Because of this expressiveness, the actual constructors of the LEGO type `thm` are visible, and cannot be misused to construct non-theorems.

The LEGO notation `{x:A}B` represents the dependent function type, or universal quantification, so the type of constructor `Axiom` expresses "for any context, G, and any proposition, a, if $a \in G$ (where membership is with respect to the decidable equality `prop_eq`) then judgement $G \vdash a$ is justified by AXIOM.

One further point slightly confuses our comparison of LCF with the implementation of our example in LEGO; since LEGO supports refinement proof directly, we don't need the notion of tactic; the validations (i.e. inference rules) are programmed directly, and LEGO manages their use for refinement.

Derivable Rules The forgetting tactic is proved

```
Goal forget : {G|context}{a,b|prop}(thm G b)->thm G (arr a b);
intros ___;
Refine [H:thm G b]ELim (Intro (Intro (Axiom ?))) H;       (** main step **)
Equiv is_tt (orr (prop_eq ??) (orr (prop_eq ??) (member ???)));
Qrepl prop_eq_refl b; Qrepl tt_orr_zero (prop_eq b a); Refine Q_refl;
```

The main step is very similar to the SML validation. The rest of the proof is manipulation of boolean equalities, which is built-in to SML. This tactic is really a function, and has the same kind of behavior as the SML `forget_tac`: it doesn't look inside the proof of `(thm G b)`, and if normalized, constructs a new proof by gluing parts together. However, it does not have to be normalized to be used for refining a goal, as it cannot fail and is explicit about the theorem it proves.

Admissible rules What about the weakening tactic? We define $\subseteq$,

```
[subCxt [G,H:context] =
   {b:prop}(is_tt (member prop_eq b G))->is_tt (member prop_eq b H)];
```

and prove weakening:

```
Goal weaken : {K,G|context}(subCxt K G)->{a|prop}(thm K a)->thm G a;
Claim {K|context}{a|prop}(thm K a)->{G|context}(subCxt K G)->thm G a;
intros; Refine ?1; Immed;
Refine thm_elim [K:context][a:prop][_:thm K a]            (** main step **)
                {G|context}{sC:subCxt K G}thm G a;
(* axiom *)
intros; Refine Axiom; Refine sC; Immed;
(* intro *)
intros; Refine Intro (J_ih ?); Expand subCxt;
intros; Refine fst (orr_character ? (member prop_eq b1 G)) H;
intros eqb1a; Refine snd (orr_character ??); Refine inl eqb1a;
intros mem; Refine member_tl_lem; Refine sC; Immed;
(* elim  *)
intros; Refine ELim (J_ih ?) (K_ih ?); Immed;
```

There is much in this proof script that is unexplained, but I wanted to compare the size of this proof with the size of `weaken_tac` (section 3). In this proof, the

first two lines rearrange the goal to set up for the main step, which is induction (`thm_elim`) on the structure of the given object of type (`thm K a`). This main step is analagous to the main step of the validation returned by `weaken_tac`, namely (`case destr_thm th of ...`), case analysis on the structure of the given `thm` object. The three cases in the proof are marked by comments: `axiom` is the base case, and `intro` and `elim` are the induction cases. Where the tactic calls itself recursively, the LEGO proof uses induction hypotheses, called `J_ih` and `K_ih` in the script. This function also behaves like the SML validation: if normalized it actually traverses the proof of (`Thm K a`) and builds a new derivation of the same size. However, this proof doesn't have to be normalized to be used to refine a goal.

Decision procedures Finally we have a decision procedure for WEAKENING. You are free to prove (`subCxt K G`) however you like: using a program that does arbitrary search (and may fail), step by step directed by the user (who may fail to construct the desired proof), or some combination; this is the "decision" part. Once a proof, call it `p`, of (`subCxt K G`) is found, however, it justifies admissibility of the rule:

$$\frac{K \vdash a}{G \vdash a}$$

which can be used without further computation. That is, (`weaken p`) is a non-canonical proof notation in the formal meta theory, ECC, that replaces a goal (`thm G a`) by a goal (`thm K a`), for any `a`. This notation could be normalized to a canonical proof notation if desired, but need not be to safely construct object proofs.

5 Concluding Remarks

5.1 Is it necessary at all?

HOL [Gor88], using pure LCF style, is one of the most successful proof tools in the world. Do HOL users feel the need to have more tactics (section 3), or tactics that don't have to be executed (section 4)? I know of no call for either of these features; in fact one serious user has written a paper on extensibility [Har95b] in which he doesn't mention that HOL can not represent tactics for admissible rules that are not derivable, and argues passionately against the need for tactics that don't have to be executed (although he doesn't consider the idea I am proposing in this paper).

On the other hand, the Nuprl group has put considerable effort into avoiding the execution of tactics, using ideas similar to this paper [KC86, Kno87] as well as various notions of *partial reflection* and *reflection* [KC86, How88, ACHA90]. They seem to believe it is an important issue.

5.2 What properties must a meta language have?

I have emphasised that a meta system allowing non-canonical objects to be treated as indirect proof notations for a represented object system must have total functions, and be expressive enough to specify which judgement a `thm` object proves. In section 4 I used ECC with inductive types as such a meta language. The choice of meta language involves finding a balance of proof-theoretic strength and expressiveness.

Types are not required. Some untyped logics of total functions, and inductively defined relations, are suitable for such a meta language. I have in mind Feferman's system FS_0 [Fef88], which is designed expressly as a meta language for formal mathematics.

Inductively defined relations are required. The Edinburgh Logical Framework (LF) [HHP92] is a logic of total functions and first order dependent types. However, lacking *internally definable* inductive types, it is not adequate as a meta language to be used as I am suggesting. Similarly, other frameworks such as Isabelle and λ-Prolog [Fel89], which are proof theoretically stronger than LF but also lack inductively defined relations, are not suitable. These three frameworks can express the syntactical constructors of such things as terms, formulas and proofs, as "introduction rules", but they lack a notion of "inductiveness" necessary to have the corresponding elimination rules. Pfenning, working with his ELF implementation of Edinburgh LF, has been experimenting with using another level of meta language, above LF, to represent inductive closure [PR92].

Proof theoretic strength. I have said that the meta language must have total functions and inductively defined relations relations. However, different meta languages have different inductive strengths, and different sets of total functions. For example, in ECC it is possible to write a sound and complete typechecking tactic for system F, while in FS_0 this is not possible, as system F is stronger than FS_0. However it is possible in FS_0 to write a sound typechecking tactic for system F which succeeds whenever the ECC tactic would have succeeded in reasonable time. There is also the question of whether the best program in FS_0 for some function is infeasibly worse than a program in ECC for the same function (e.g. because FS_0 lacks generalized induction which ECC has); I do not think the answer to this is known.

5.3 Can validations actually be programmed in such a language?

How can total functions be used as validations? I am suggesting a meta language of total functions, but it seems that non-termination and the raising of exceptions is essential in LCF, both for fundamental reasons and for convenience in programming tactics. In fact, the compositionality of tactics using tacticals such as ORELSE and THEN, which is based on the use of exceptions, is one of the most succesful aspects of LCF style. Nonetheless, I claim that exceptions

and non-termination are not essential for LCF, or for similar compositionality of tactics in a meta language of total functions.

First notice, as an aside, that one use of exceptions in LCF, the need for the forward proof constructors to fail, does not occur when using a sufficiently expressive meta language. For example, in LCF there is no way to express the side condition of rule AXIOM, that $a \in G$, in the typing of forward proof constructor `Axiom`, so `Axiom` may discover it has to fail. In the ECC representation, however, the type of `Axiom` shows that it cannot be called without being given a proof that $a \in G$.

More fundamentally, the style of programming with exceptions can be emulated in pure functional languages by raising each type to have an error value [Spi90] or using the monad of exceptions [Wad92]. Whether these approaches can be as efficient and convenient as conventional exceptions I do not know, but their proponents claim they are very usable in practice. Similarly, programming with general recursion can be simulated in languages of total functions; raise each type to have a "not terminated" value, and return this value if the computation hasn't terminated in some parametrically determined number of steps. There is some discussion of this in [Pol95].

Can such validations be executed efficiently enough? First notice that this question is somewhat beside the point for the technique I am suggesting, as our goal is to avoid actually normalizing the non canonical object proofs represented by our validations; the SML version of `weaken_tac` is much more efficient than the ECC proof `weaken`, but the proof doesn't have to be executed in order to refine a goal, while `weaken_tac` does. Nonetheless, some computation, such as checking side conditions, does have to carried out. Further, eficiency of computation in the meta language is important for the use of partial reflection in conjunction with this approach.

The current implementation of LEGO executes its object level computations very slowly. These computations are not compiled or optimised, and they involve reducing a whole constructive proof, rather than just its computational content. However, there is much work on extracting (total) functions from constructive proof (for example [Con86, HN88, PM89, DFH+91] and work of Berardi and his collaborators), and their is no reason why such extracted programs can not be compiled and executed efficiently.

Remark. In order to use a program extracted from a proof that a rule is admissible as a non canonical proof notation for that rule, we need programs extracted from proofs to be objects of our meta language, and to have types (or other properties) showing what rule they compute. The program extraction of Nuprl does meet these requirements; the program extraction currently implemented in Coq [DFH+91] does not, as programs are extracted into a language that is external to the Coq object calculus. I plead with workers in this field to address our need to use extracted programs for formal as well as informal calculation.

5.4 Related work

Most of the work on extensibility of proof checkers is descended from the LCF notion of tactics, as is the suggestion in this paper. The idea to use a meta language with inductively defined relations appears in work of Knoblock [Kno87], who defines a formal meta language for Nuprl, and in the work of Matthews [MSB93], who uses FS_0 as a meta language.

There is one other thread of related work, dealing with various notions of reflecting a logic into itself. For an excellent survey of this approach, and an empassioned plea against it, see [Har95b]. A well known use of one form of reflection is in the Boyer-Moore system [BM88], NQTHM, where a notion of `META` rules allows user written functions operating on representations of terms to be proved correct with respect to a meaning function on representations, and used for simplification rewriting directly by the proof system.

The Nuprl group have been working for some years on various notions of reflection [KC86, How88, ACHA90, ACU] as a means to have tactics that don't have to be executed. They suggest using Nuprl as a meta language for a (quoted representation of) Nuprl, but including a new rule justifying a proof of a judgement by a quoted proof for the quotation of that judgement. Whereas my suggestion in this paper gives a safe and extensible implementation of almost any formal system, the Nuprl suggestion is for a special logic with a very strong reflection meta rule. What they hope to gain by this is not just tactics in some meta language that don't have to be executed, but the ability to use already proved object language theorems in defining these tactics and proving their correctness. This is a very reasonable point, as serious mathematics will be necessary to prove correctness of interesting decision procedures. It is unreasonable to develop this material twice, once in the meta system and once in the object system. Even worse, without reflection we will eventually have to write some tactics without the benefit of another meta level. Nonetheless, reflection seems very heavy. Since no serious examples of Nuprl reflection have yet been completely formalized (but see [ACU]) it remains to see if the claimed benefits are forthcoming. The same can be said of my suggestion in this paper.

5.5 Conclusion

My suggestion is little different from LCF, just replacing one computational meta language (ML) with another (ECC, FS_0,...). The philosophical point is that it is then possible to accept non canonical proof notations as object level proofs, removing the need to actually normalize them. There are problems to be worked out in practice, such as extraction of programs from constructive proof, and efficient execution of pure, total programs. Although this approach doesn't address the difficulty of proving correctness of tactics in the meta level, it is immediatly useful for tactics with structural justification (e.g. weakening) which are not even representable in LCF, and are infeasible in the Nuprl variant of LCF. Since it can be used for any object system without adding new principles such as reflection, and is compatible with other approaches to extensibility (especially

partial reflection), it should be considered as part of the answer to extensibility in proof checkers.

References

[ACHA90] Allen, Constable, Howe, and Aitken. The semantics of reflected proof. In *LICS Proceedings*. IEEE, 1990.

[ACU] William Aitkin, Robert Constable, and Judith Underwood. Metalogical frameworks II: Using reflected decision procedures. Technical report, Cornell University. To appear.

[Avr91] Arnon Avron. Simple consequence relations. *Information and Computation*, 92:105–139, 1991.

[BM81] Robert S. Boyer and J S. Moore. Metafunctions: Proving them correct and using them efficiently as new proof procedures. In Robert S. Boyer and J S. Moore, editors, *The Correctness Problem in Computer Science*, pages 103–184. Academic Press, New York, 1981.

[BM88] Robert S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[Con86] Robert L. Constable, *et. al. Implementing Mathematics with the Nuprl Proof Development System*. Prentice–Hall, Englewood Cliffs, NJ, 1986.

[dB85] Nicolas G. de Bruijn. Generalizing automath by means of a lambda-typed lambda calculus. In *Proceedings of the Maryland 1984-1985 Special Year in Mathematical Logic and Theoretical Computer Science*, 1985.

[DFH+91] Dowek, Felty, Herbelin, Huet, Paulin-Mohring, and Werner. The Coq proof assistant user's guide, version 5.6. Technical Report 134, INRIA-Rocquencourt, December 1991.

[Fef88] Solomon Feferman. Finitary inductively presented logics. In *Logic Colloquium '88, Padova*. August 1988.

[Fel89] Amy P. Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language*. PhD thesis, University of Pennsylvania, September 1989. MS-CIS-89-53.

[FFM90] Mick Francis, Simon Finn, and Ellie Mayger. Reference manual for the Lambda system. Technical report, Abstract Hardware Limited, 1990.

[GMW79] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*. Springer-Verlag, 1979.

[Gor88] Michael Gordon. HOL: A proof generating system for higher-order logic. In Birtwistle and Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, 1988.

[Har95a] John Harrison. Binary decision diagrams as a HOL derived rule. *The Computer Journal*, 38(5), 1995.

[Har95b] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, UK, 1995. See `http://www.cl.cam.ac.uk/ftp/hvg/papers/`

[HHP92] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1992. Preliminary version in LICS'87.

[HN88] Susumu Hayashi and Hiroshi Nakano. *PX: A Computational Logic*. MIT Press, 1988.

[How88] Douglas Howe. *Automating Reasoning in an Implementation of Constructive Type Theory.* PhD thesis, Cornell University, June 1988.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus*, volume 1 of *London Mathematical Society Student Texts.* Cambridge University Press, 1986.

[KC86] T. Knoblock and R. Constable. Formalized metareasoning in type theory. In *LICS Proceedings.* IEEE, 1986.

[Kle67] Stephen C. Kleene. *Mathematical Logic.* Wiley, New York, 1967.

[Kno87] Todd Knoblock. *Metamathematical Extensibility in Type Theory.* PhD thesis, Corness University, December 1987. Technical Report 87–892.

[LP92] Zhaohui Luo and Robert Pollack. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, LFCS, Computer Science Dept., University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, May 1992. Updated version. See `http://www.dcs.ed.ac.uk/packages/lego/`

[Luo94] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science.* International Series of Monographs on Computer Science. Oxford University Press, 1994.

[Mar85] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. Technical Report 2, Scuola di Specializzazione in Logica Matematica, Dipartimento di Matematica, Università di Siena, 1985.

[Mil92] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14:321–358, 1992.

[MP93] James McKinna and Robert Pollack. Pure Type Sytems formalized. In M.Bezem and J.F.Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications, TLCA '93*, pages 289–305. Springer-Verlag, LNCS 664, March 1993.

[MSB93] Sean Matthews, Alan Smaill, and David Basin. Experience with FS_0 as a framework theory. In G. Huet and G.D. Plotkin, editors, *Logical Environments.* Cambridge University Press, 1993. Formal Proceedings of the Second Workshop on Logical Frameworks, Edinburgh, May 1991.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[Pau93] Lawrence C. Paulson. Introduction to isabelle. Technical Report 280, University of Cambridge, Computer Laboratory, 1993. See `http://www.cl.cam.ac.uk/Research/HVG/isabelle.html`

[PM89] Christine Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In Association for Computing Machinery, editor, *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, 1989.

[Pol94] Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1994. Available by anonymous ftp from `ftp.cs.chalmers.se` in directory `pub/users/pollack`.

[Pol95] Robert Pollack. A verified typechecker. In *TLCA '95, Proceedings of the Second International Conference on Typed Lambda Calculi and Applications, Edinburgh.* Springer-Verlag, LNCS, April 1995.

[PR92] Frank Pfenning and Ekkehard Rohwedder. Implementing the meta-theory of inductive systems. In D. Kapur, editor, *Proceedings of the Eleventh An-*

nual Conference on Automated Deduction, Saratoga Springs, New York, number 607 in LNAI, pages 537–551. Springer-Verlag, June 1992.

[Spi90] Mike Spivey. A functional theory of exceptions. *Science of Computer Programming*, 14:25–42, 1990. North Holland.

[Wad92] Philip Wadler. The essence of functional programming. In *Nineteenth Annual Symposium on Principles of Programming Languages, Santa Fe, New Mexico*, January 1992.