

Desiderata for Interactive Verification Systems

DRAFT

Konrad Slind and Christian Prehofer

October 25, 1994

Abstract

What facilities should an interactive verification system provide? We take the pragmatic view that the particular logic underlying a proof system is not as important as the support that is provided. Although a plethora of logics have been implemented, we think that there is a common kernel of support that a proof system ought to provide. Towards this end, we give detailed suggestions for verification support in three major areas: *formalization*, *proof*, and *interface*. Although our perspective comes from experience with highly expressive logics such as set theory, higher order logic, and type theory, we think our analyses apply more generally.

Introduction

Currently, theorem provers are used in the verification of both hardware and software [GM93, ORS92, BM90, HRS90, FFMH92], the formalization of informal mathematical proofs [FGT90, CH85, Pau90b], the teaching of logic [AMC84], and as tools of mathematical and metamathematical research [WWM⁺90, CAB⁺86].¹

In this paper we describe important facilities that one might want to find in a theorem prover. Our perspective comes from using proof systems for the verification of computer systems. Verification potentially requires sophisticated mathematics and so a versatile verification system ought to be usable in principle as a theorem prover in almost any mathematical domain. In this paper, we will largely regard the logic as given, although some of our biases will surface, and devote ourselves mainly to the question of what tools an implementation ought to supply. To give some form to the discussion, we split the usage of theorem provers into three main areas:

- Formalization, the capture of a domain of interest in the logic. In mathematics finding the right definitions and notation is central, and this is equally true when formalizing computer systems and their properties.

¹This assignment of theorem provers to applications is fairly loose and certainly not exhaustive. We will occasionally mention particular theorem provers in various contexts where the facilities they provide are noteworthy. Again, this will not be exhaustive and we apologize for our many oversights.

- Proof, the use of the rules of the logic to establish facts about the formalized domain.
- Interface, the provision of tools so the user can interact at a high level with formalizations and proofs.

Integrating the wide variety of useful formalization and proof methods into a compact and manageable system is a challenging task. The primary needs in interactive verification are expressiveness in formalization, efficiency of the *user's* time in proof, and support for large, evolving objects, namely proofs and theories. Just as in programming, some tension arises from the twin requirements of expressiveness and efficiency. For many verifications, the expressive power of higher-order constructs is important for natural and concise representations, which is crucial for interactive proofs. Even when the theories involved are first-order, a need for higher-order reasoning may arise from tools needed to structure large proofs, as discussed below. On the other hand, one might hope that sophisticated higher order implementations could provide as much efficiency as the setting warrants, *i.e.*, approximately “first order” speeds when the prover is in an essentially first order setting. This worthwhile goal must be weighed against the desire that the implementation be small enough that its correctness can be shown.

1 Formalization

This activity extends from the identification of a domain of enquiry to the establishment of definitions (or axioms) modelling that domain. This process is a mostly *informal* and exploratory activity, hence feedback is important to ensure that definitions (or axioms) do in fact capture the domain of enquiry. Currently, most verification systems give poor support for this task², and checking the accuracy of definitions is usually done by managing to prove desired theorems. This often fails, and so the formalization process goes in a “define—attempt proof—revise definition” cycle analogous to the “compile—run—debug” cycle of programming.

In the rest of this section, we consider various facilities that are of use in the formalization process: type systems, definition facilities, theories, definition re-use mechanisms, and different approaches to verification theories.

1.1 Types

Type systems are of fundamental importance since they offer the principal means for saying *what* the domain of interest is. There is a great multitude of type systems[CW85], ranging from no types, as in set theory, to first order types with unsorted, multisorted, and polymorphic variants, to lambda-calculus based type theory, possibly with subtypes and

²although see [ORS92], where the problem is at least taken seriously and [BM90] where logic functions can be executed, thus giving a testing facility

dependent types thrown in. Perhaps the central issue in choosing a type system is that the expressiveness of the type system may impede efficient or even effective implementation, since such issues as the decidability of type inference get raised. Our experience shows that advanced type systems are vital for compact formalization and are worth some extra implementation effort. Out of the range of type facilities, the following stand out:

- Polymorphism is a very useful tool for concise and re-usable formalization, but it has an implementation cost. Furthermore, the extra degree of freedom it implies can give difficulties when extending inference methods from monomorphic to polymorphic status. In spite of that, the ability to do type inference in languages allowing ML-style polymorphism[Mil78] is a tremendous help in formalization, since ML type inference gives a quick check for consistency of term and formula construction. Nearly always, the type inference problem for polymorphic type systems with more expressive types than ML types is undecidable[CW85].
- For some tasks dependent types are useful. An example is modelling programming language functions with a variable number of arguments, which occurred in our compiler verification case study. The compiler needed to be able to compile functions of arbitrary arity. Informally, a function f on integers which takes n arguments could be modelled with the dependent type $f : \text{tuple}(n)(\text{int}) \rightarrow \text{int}$. Notice that the type depends on a term n . Unfortunately, type inference with dependent types is undecidable. A lot of applications of dependent types can be handled with a combination of special syntax for bounded quantification and specialized proof support, as in Isabelle.
- Subtypes provide a way to invest a type with more meaning while retaining the same operations. For example, the integers without zero are a subtype of the integers that supplies all the usual integer operations as well as providing division. Subtypes are often used to develop domains “top-down”, *e.g.*, starting from an initial type of “number” with basic operations, one can build, in succession, the reals, the rationals, the integers, and finally the natural numbers. In this approach, when an operation is applied to an item in a subtype, the item often needs to be coerced to an “ancestor” type in order for the operation to be performed, which has implications for type inference and proof.

1.2 Definitions

Avoiding inconsistency in an axiomatization is very hard, as such notable logicians as Martin-Löf, Church, and Frege can attest. Although these worthies were axiomatizing *logics*, the issue of consistency is equally problematic when building a formalization *inside* a logic. Therefore, it is essential that *principles of definition* exist in the system. Such principles conservatively extend the logic, and thus faithful users of definitions are guaranteed not to introduce any inconsistency not present in the original logic. This methodology requires a bit more work than the arbitrary assertion of axioms, but because of its safety, it is to be preferred. (Of course, one may at times need to assert

an axiom, if one wants to work in various flavours of set theory, for instance.) A good collection of definition facilities would include the following:

- subtype definition, *e.g.*, defining the integers as a particular subset of $num \times num$. This is not the same as the subtypes found in the “Types” section since the new type does not inherit any operations from the type it is derived from. In fact, subtype definition is a “bottom-up” approach, in contrast to the “top-down” approach of a system offering subtypes.
- concrete type definition, *e.g.*, $tri = A \mid B \mid C$ of num defines a type tri with three constructors: $A : tri$, $B : tri$, and $C : num \rightarrow tri$.
- recursive type definition, *e.g.*, $btree = Leaf\ of\ \alpha \mid Node\ of\ btree \times \alpha \times btree$ defines a type of polymorphic binary trees.
- simple abbreviation of closed terms, *e.g.*, $K = \lambda xy.x$.
- function definition using ML-style pattern matching, *e.g.*, $(f\ A = 12) \wedge (f\ B = 13) \wedge (f\ (Cn) = n^2)$ would define a (nonsensical) function f of type $tri \rightarrow num$, where tri is as defined above.
- primitive recursive function definition, *e.g.*, $(A+0 = A) \wedge (A+(Sn) = S(A+n))$ could be used to define addition in Peano arithmetic. Higher order primitive recursion can define more functions than mere primitive recursion over the natural numbers, but such definitions do not directly express the “real” recursion.
- definition by well-founded recursion, *e.g.*, $(gcd\ a\ 0 = a) \wedge (gcd\ x\ (Sy) = gcd\ (Sy)\ (x\ mod\ (Sy)))$ could be used to define the greatest common divisor algorithm. Well-founded recursion subsumes primitive recursion, and can be used to define any terminating functional program. A problem with well-founded recursion is that the user has to show that recursive calls are on decreasing arguments, which means that not all definitions can be checked automatically.
- inductively defined sets, *e.g.*, $(0 \in num) \wedge (n \in num \supset (Sn \in num))$ could be used to define the set of natural numbers, as a very simple example.
- mutual versions of both inductive and recursive definitions, *e.g.*, the syntax of many programming languages can only be defined by a mutual recursion.

Complex principles of definition such as those for recursive definitions are often just hardwired into a theorem prover implementation. This methodology is vulnerable to the same criticism as the assertion of axioms: it is unsafe. Also, it requires a larger effort to ensure, formally or not, that the implementation of the logic is correct. An alternative is to *derive* them from a primitive principle of definition as exemplified in the HOL system[GM93]. Another alternative is meta-theoretic extensibility which is discussed below under the name of computational reflection.

Some case studies in verifying functional programs [AL93] show that structuring a program in small units results in properties that are more compact to state and easier to prove. Therefore, a design style that emphasizes definitions composed from small subcomponents is not only useful from the vantage point of software engineering, but the verification may also benefit. For a dissenting view, see [Lam93a].

Partial functions

In many formalizations, a notion of partial function can be useful. For instance, the definition of *general recursive* functions is of interest to those involved in software verification, *e.g.*, [BDD⁺92], but this feature is usually not easy to provide since most theorem provers to date are based on logics of total functions (exceptions are IMPS[FGT90] and LCF[Pau87]). Of course, domain theory[Sco93] can be formalized in a logic of total functions, but the two notions of function in such systems have not to date been merged smoothly. Logics of partial functions are numerous[CJ90, Kup93] but the major obstacle in their use is the fact that all reasoning algorithms in such logics need to be able to deal with partiality, which can be a major implementation obstacle.

Stopping short of domain theory or a special logic for partial functions, there are several approaches that we can see to partial functions in a logic of total functions: one can make the domain of the function smaller; one can make the range larger; or one can partially specify the function:

- Subsetting the domain of the function gives rise to a relational style of specification: in higher-order logic one may define, once and for all, the filtration $\mathcal{F}(P, f, i, y)$ of function f by predicate P on the *argument* of f as $Pi \supset (fi = y)$. Similarly, the filtration $\mathcal{F}(P, f, i, y)$ of f by P on the *result* of f is $P(fi) \supset (fi = y)$. The relational style interacts usefully with subtyping or dependent types in systems that provide such support.
- Extending the range of the function by adding a value that stands for "undefined" allows a functional style, at the cost of having to support the "undefined" value in some way. One can use the *option* type, which is a common functional programming technique for "totalizing" partial functions. The *option* datatype would be defined in an ML style as

$$\alpha \text{ option} = \text{None} \mid \text{Some of } \alpha.$$

Under this approach, a partial function f of type $\alpha \rightarrow \beta$ would be mapped into a total function f' of type $\alpha \rightarrow \beta \text{ option}$.

- A principle of *weak specification*[GM93] allows the definition of a function on only part of its domain. For instance, a weak specification of the function that takes the head of a list is just $hd(h :: t) = h$: the clause for the empty list is left out. (Loosely, the only thing that can be proved about $hd[]$ is that it is identical to itself.)

1.3 Theories

Many proof systems implement a means of bundling related types, constants, definitions, axioms, and theorems into a single entity known as a *theory*. Theories are often arranged hierarchically and used to structure proof efforts. The major operations on theories are: creating a theory, establishing and maintaining dependencies between theories, and accessing a theory so as to use its contents for formalization or proof[Pau87].³ The following are several pragmatic considerations in the implementation and use of theories:

- The proper management of a large hierarchy of theories requires the implementation of persistence, *i.e.*, being able to write a theory out to disk and read it back in safely. If the implementation language of the theorem prover supports persistence, then all is well, or ought to be; otherwise, quite a lot of theory-handling code will need to be implemented. One property that has major impact on memory usage is that the importing and exporting functions should preserve *sharing* of subterms.
- Maintaining consistency in the theory hierarchy is important. For example, when a theory changes all its descendants ought to be marked so that theorems from them cannot be used until revalidated.
- An operation analogous to *paging* in operating systems can be helpful when the theory hierarchy grows so large that the theories in memory take up space that reasoning procedures could profitably use. Only the theories that are necessary to the proof at hand need be resident (and a refinement of this keeps resident only the definitions and theorems that are actually used in the proof).

1.4 Reuse

Once good definitions and theories have been established, they ought to be re-used. Reuse facilities for logics are very much like those for programming. There are several not necessarily incompatible (so far as we know) approaches:

- Polymorphism. Polymorphic definitions may be instantiated to any type. This provides a great economy in theory development, since for example, a theory of polymorphic lists means that separate theories for integer lists and string lists won't have to be developed. Type theories whose terms allow ML-style polymorphism [GM93, Pau90b, FFMH92] can have the type of terms automatically inferred, which is of tremendous help in the formalization process as previously mentioned.
- Abstract theories. Just as polymorphic definitions give re-usability at the term level, a notion of *parameterized* theory gives reusability at the theory level. A parameterized theory T can be thought of as having fixed but unspecified types, fixed but unspecified constants (over these types), and axioms asserted in terms of

³This conception of theories is closely related to facilities for specification-in-the-large[Wir90]

these constants. Any theorems proved from the axioms of T are called “abstract theorems”. T is instantiated by providing a theory S that has types, constants, and theorems such that it can be seen as a kind of *substitution instance* of T . That is, there are substitutions for the constants and types of T such that every instantiated axiom of T is a theorem of S . If that is true, it is valid to instantiate the abstract theorems of T ; these become theorems of S . The Imps theorem prover makes extensive use of abstract theories [FGT90]. Theorem provers that do not offer polymorphic types sometimes attempt to recover a flavour of polymorphism by offering parameterized theories, as in PVS.

- **Constrained polymorphism.** This is a development in Isabelle arising from advances in the problem of combining parametric polymorphism with overloading in functional programming languages [WB89, Nip93b]. It can be viewed as an intermediate point between polymorphism and abstract theories. Briefly, an Isabelle abstract theory uses sorts on type variables to implement the notion of “fixed but unspecified constant”. This is then used to implement standard abstract theories. One nice aspect of this approach is that abstract theory hierarchies can be expressed via the subsort relation.

1.5 Verification theories

Many formalisms have been proposed for verification purposes, *e.g.*, Floyd-Hoare logic [Hoa69], LCF [Sco93, GMW79], IO automata [LT89], the Bird-Meertens calculus [BW88], *etc.* There is a continuum of ways such verification theories can be supported in a theorem prover; we identify just a few:

- **No support.** The basic devices of the logic are used to model the entity being verified. The most successful use of this is probably Gordon’s approach to hardware verification [Gor85], where circuits are modelled by relations, *i.e.* predicates, and wiring is done by relational composition, *i.e.*, existential quantification. This lightweight approach has the advantage that no specialized reasoning tools are needed to perform verifications.
- **Semantical support.** The semantics of the verification theory is defined in the logic, the implementation to be verified is defined in terms of those semantic constructs, and so the verification is carried out in the semantics. In some cases, verification conditions can be derived from the semantic embedding; the embedding ensures that the verification conditions are consistent with the original correctness statement. For instance, the KIV system provides a formal relation between a specification and a program module, based on a *signature mapping* [HRS90]. Such support means a large gain in security as the proof obligations can be generated automatically thus freeing the user from finding them [ORSvH93, Bro92]).
- **Syntactical support.** The syntax and semantics of the verification theory is defined in the logic. This allows the proof of *meta-theoretic* results about the formalism, typ-

ically by inductive reasoning over the syntax. Formalizing the meta theory requires an expressive logical system, often set theory or higher-order logic. For instance, the admissibility of predicates for computational or fixed point induction was not definable in the first order logic of Cambridge LCF[Pau87]: an ML function was used to give an incomplete check of predicates for admissibility by examining their structure. A syntactical support approach that improves on this can be found in HOLCF[Reg94].

Conclusion

Strong facilities for formalization are essential in verification. In our case studies, we found the expressiveness of higher-order logic or set theory to be a tremendous help in formalization. In general, the more support a logic provides for formalization, the more complex the logic is to implement; however, one can have a relatively simple logic and still have practically useful formalization tools.

2 Proof

As mentioned before, we view the proof process as “coroutining” with formalization until the definitions are found to adequately capture the domain of interest. After that, the activity of doing proof is the focus. Some relevant properties of proof are the following:

- As approximately forty years of research into automated reasoning has shown, the intractability of almost all proof procedures does not mean that useful work cannot be done with them.
- Proofs can grow to a huge size, and it is a real problem to display the important information in a proof in an understandable format.
- Interactive proofs are produced incrementally, which has implications for the kinds of proof procedures that are useful. For instance, procedures that transform the proof into a low-level representation such as clauses are not useful at intermediate steps. (They have, however, proved very useful when applied as concluding steps in interactive proofs, as shown in our case studies.)
- Proofs have to meet a range of sometimes conflicting criteria among which are: efficiency, elegance, readability, robustness under change of proof procedures, reusability, *etc.* For example, there are (at least) two types of users: those who are interested in the proof as an object of study (typically mathematicians or logicians), and those who are primarily interested in getting the theorem by whatever means possible (those involved in a large verification, for example).

With these general observations in mind, we now turn to some useful facilities for proof. First we attend to questions of style and granularity; then we describe some of the issues surrounding decision procedures; then we turn to the important area of equational proof methods; then we have a short discussion on utilizing logic programming techniques in proof; then we describe useful proof management facilities; and we conclude with a discussion of efficiency issues.

2.1 Proof style

Backwards, or goal-oriented, proof is successfully supported by Milner’s seminal idea of *tactic*[Mil85]. Almost all interactive verification systems use some flavour of tactics, and some logics are even expressed wholly in terms of backwards proof[CAB⁺86]. One of the reasons for the success of tactics is that they almost never require complicated programming: *tacticals* make it simple for users to build compound tactics. Forward proof is also quite an important method[McA89], which is not surprising, since many logics in interactive theorem provers are based on Natural Deduction, which was originally designed to model how humans do proof. A recent paper proposing an interactive method based on forward proof is [Lam93b].

For an interactive theorem prover, strong and incremental automated proof support is necessary to avoid wasting the user’s time on easily proved theorems. For verification purposes, the user should need to give only very abstract input to the prover, such as “do rewriting”, or “use decision procedures”. For instance, the Boyer-Moore [BM90] system is designed for automatic proof without any input from the user, except for providing some “hints”. However, in our case studies we found that finer control is needed at many steps. Now the problem is that in many proof systems there is a wide gap between the automatic facilities and the low-level stepwise facilities: for lack of a middle ground, the user is too often forced to work at an unnaturally low level. The following different levels of user control can be distinguished, and ought to be supported:

1. In the first refinement, the user gives the prover hints on *what* to use, *e.g.*, suggesting certain rewrite rules, lemmata or proof strategies.
2. The user sometimes wants to have even more control over *where* to use a tactic. For instance, one might want to apply simplification only to a particular premise of the goal.
3. Even more control can be exercised specifying *how* a step should be done, *e.g.*, by providing explicit substitutions for instantiating a lemma.

In some cases, users knowing the proof in detail still insist on using low-level proof methods. At difficult proof steps, where automatic methods fail, low-level support is needed and sometimes may be faster (in user and system time) than attempting higher-level support. However, higher-level proofs are clearly desirable, in particular for reusing proofs.

In spite of the success of mechanical methods of proof (backwards, forwards, and automatic), the resulting proofs are generally not readable, so a more “human-oriented” language in which to express proofs (a so-called *proof vernacular*) is still required to bridge the gap between machine and paper-and-pencil proofs[Dow90]. The ideal would be if the vernacular could be mechanically translated into prover commands, since the vernacular could then serve as a *lingua franca* for communicating proofs between humans and machines.

2.2 Decision procedures

In general, the validity of logical formulas is not decidable; however, there are some domains that do allow the decision problem to be solved. The attraction of having a decision procedure for a domain is that, in principle, reasoning in the domain can be totally automated. However, most decision procedures have fundamentally bad computational complexity. That is inescapable: at least working in an interactive theorem prover allows the use of human ingenuity to keep the complexity of formulas low. Some practically useful decision procedures are:

- Propositional logic. This is often used as a “base case” proof procedure by more powerful methods, *e.g.*, those that work by elimination of quantifiers.
- Presburger arithmetic. The (first order) theory of arithmetic involving only addition (or alternatively multiplication) is decidable[BJ89], but the decision procedure is very slow; however, for the subcase of unquantified Presburger arithmetic, effective procedures are available[NO79, Sho79]. These are very useful in the verification of hardware and software.
- Congruence closure. The unquantified theory of equality given a set of equations is efficiently decidable using the method of congruence closure[NO80]. This is useful in more cases than at first sight, since equations featuring terms with free variables can be treated as ground provided the variables are not instantiated. Using congruence closure, the first order theory of lists can be efficiently decided.
- Recently, the complexity of decision procedures for the first order theory of the reals has been improved[GV88, Gri88, HRS93], and we anticipate that efficient implementations of these will spark verifications of hybrid systems, where real world, *i.e.*, continuous, inputs must be mapped to discrete domains.
- Decision procedures for some modal logics, such as computation tree logic, or the modal μ calculus, are quite helpful in some verifications[Bry86, CES86, CPS89], but their efficiency seems to depend on clever datastructures that are hard to emulate in a general-purpose theorem prover.
- The Knuth-Bendix completion procedure[KB70, PS81] can at times give a means of deciding equality by means of rewriting. It can also be used as a semi-decision

procedure[Hue81]. The essential problem limiting the usefulness of this method is finding term orderings[Der87]. A higher-order version of completion has yet to be developed.

The combination of decision procedures is an important structuring technique. An *equality propagation* algorithm is used to combine decision procedures for the quantifier-free theories of equality, lists, arrays, and Presburger arithmetic into very useful decision procedures for program verification[NO79, Sho84].

Decision procedures are often “builtin” as basic rules of inference. This is justified using a specious argument that goes, “Seeing as this algorithm is so complicated, we had better implement it in the kernel of the logic so that it will go fast enough to be useful”. Of course, the problem is that complicated algorithms are hard to get right, and so the chance that the theorem prover is faulty is higher than if the kernel was kept small.

Decision procedures are likely to make the class of users who “just want the theorem to be proved” very happy, since, if they can phrase their problem in the decision procedure’s sublanguage and if the decision procedure doesn’t take too long on their formula, then they won’t have to think about the proof. On the other hand, users who are primarily interested in the proof may want to examine the proof in detail. When the decision procedure works by reducing the formula into a particular canonical form, it can be very hard to present its proof in a human-friendly fashion[Mil84].

When providing proof support for a domain, not only is an oracle needed, but also a module that simplifies formulas of the domain. The notion of what it means to be *simpler* in a domain is difficult to define, and is often the crux in finding a decision procedure! Therefore, heuristic simplification is sometimes the best that can be done. A simple example is the gradeschool technique of cancellation of equal terms on both sides of an equality which can sometimes be used to simplify very complicated expressions to manageable ones. A much more powerful simplifier is the “decision procedure” of the Stanford Pascal Simplifier[NO79]. The algorithms used in constraint languages[JMSY90] can also be viewed as such simplifiers.

2.3 Equational proof

Equational proof is ubiquitous and difficult to automate effectively. The most common use of equational theorems is in *rewriting*, which is one of the most heavily used theorem proving tools. Indeed, the success of the Boyer-Moore theorem prover is largely attributable to its powerful rewriting engine. The popularity of rewriting in interactive proof is largely due to the fact that it is a user-controllable automatic method that changes the goal in a (usually) understandable way. The following seem to be useful extensions to the basic method of rewriting. Some already exist, *e.g.*, in HOL, Boyer-Moore and Isabelle, while some are, as yet, untested in practice.

- Rewriting from the current assumptions is very natural. For instance, in a case like $X = t \wedge Y = t' \vdash? P(X, Y)$ ⁴, one often wants to rewrite X and Y on the right hand side. Obviously, this should only be an option, as it is often useful to shorten formulas by writing them in this form. The situation is more complicated when local quantifiers or conditions are involved. For instance, in

$$p(f(b)), \forall x. q(x) \supset a = f(x), q(b) \vdash? p(a)$$

it is natural to rewrite $P(a)$ to $True$ with the local rules $p(b) \rightarrow True, q(b) \rightarrow True$ and $q(X) \supset a = f(X)$. Notice that this performs “on-the-fly” quantifier instantiation.

- Conditional rewriting. In order to use a conditional rewrite rule $P \supset (lhs = rhs)$, the condition P must be proven, which is often done by using the current hypotheses of the goal in a recursive invocation of rewriting. Non-termination of conditional rules is a common problem. For instance, consider a rule of the form $p(X - 1) \wedge \dots \supset p(X) \rightarrow True$, which loops when attempting to solve the condition by rewriting. Conditional rewriting is often not deterministic. When free variables in the conditions can be solved by different instantiations, we come close to logic programming or *narrowing* [Fay79]: consider

$$P(f(a)), \forall x. Q(x) \supset c = f(x), Q(b) = true, Q(a) = true \vdash? P(c)$$

where pure simplification is not sufficient and real search is needed. Or consider

$$\forall X. p(X - 1) \wedge \dots \supset p(X), p(3) \vdash? p(5),$$

where a breadth-first search may solve the goal. Narrowing can be seen as a generalization of logic programming to arbitrary conditional equations [BGM88]. Although Horn clause logic is not a complete proof strategy, together with case splits (which are commonly used to divide and conquer a proof) it is often sufficient, as our case studies suggest. Recent work on resolution theorem proving shows completeness results for a similar strategy [BF93].

- Automatic case splits are another useful tool in term rewriting. An “if” programming statement is a simple example where case splits can be recognized syntactically. Consider

$$(x \supset c) \wedge (a \supset b) \wedge (b \supset c) \vdash? \text{ IF } P \text{ THEN } a \text{ ELSE } x \text{ ENDIF } \supset c,$$

which can be solved by case splitting and term rewriting. Ideally, the user should be able to change the set of automatically recognized case splits. An example for a domain specific case split is sets, where a term $x \in (\{a\} \cup B)$ should be split into the cases $x = a$ and $x \neq a$. Notice that there is a difference between term rewriting and case splits, as a case distinction adds a new premise.

⁴We will use the notation $A \vdash? g$ to represent a goal g under assumptions A .

- Equational theories such as associativity (A) and commutativity (C) occur often and as a result there are numerous methods concerning particular equational theories [Sie84] and also general methods for arbitrary equational theories *e.g.*, narrowing [Fay79]. Various logic implementations treat equational theories differently: for instance, rewriting modulo AC is built into the Larch Prover [GG89], while the Boyer-Moore prover utilizes ordered rewriting [BM79, MN90] in order to avoid the large [KN92] number of unifiers that can be produced by AC-matching.
- Interactive quantifier instantiation is tedious and error prone, since it often means stooping to the use of low-level proof methods. An alternative might be to lift inference rules (by means of matching and unification) to work modulo quantifier equivalences. For instance, existentials that occur negatively ought to be treatable as universals by higher-level inference rules.
- Higher-order rewriting. Higher order rewriting allows one to rewrite underneath bound variables. In practice, the most successful approach to higher-order rewriting is Paulson's congruence-based rewriting [Pau83]. In the higher order setting, the theory and practice of equational reasoning has not yet been fully worked out. In particular, we want to lift useful first order methods to higher order by using higher order unification and matching. Difficulties arise because the number of matchers/unifiers is not unitary, and sometimes not finite, although see [Mil91, Nip93a, Pre93a] with regard to higher order unification.
- Higher-order rewriting modulo equational theories. Once higher-order rewriting is available, some extensions become interesting, in particular rewriting modulo equational theories [Pre93b]. For instance, we may have an abbreviation $Invar(T) = \forall i, j. P(T, i, j) \wedge Q(T, j, i)$. Then in folding a term $\forall i, j. Q(T, j, i) \wedge P(T, i, j)$, commutativity of \wedge should be considered. An example for rewriting modulo higher-order theories are fixed-point equations such as $fix(F) = F(fix(F))$, which cannot be used as rewrite rules since they will cause rewriting to loop. Such examples often occur when dealing with infinite objects, *e.g.*, recursive programs or infinite streams.
- An application of rewriting methods to reasoning over *transitive relations*, of which equality is an example, can be found in [BG93].

What about full (equational) unification, instead of just matching? In the case of higher order unification, it is used in Isabelle to apply rules and to implement derived automatic theorem proving tools. In most other interactive theorem provers, and in its full generality, this area seems to be undeveloped.

2.4 Logic programming techniques

Logic programming technology can be successfully adapted to interactive proofs. The main idea here is *logic variables*, a Prolog implementation technique. For instance, when inserting a lemma or eliminating a quantifier, many systems require the user to provide

the right instantiations or supply heuristics for doing so. A better method is to, for example, insert a lemma with new (free) logic variables and let the system find the right instances by unification and possibly backtracking. The flexibility of logic programming techniques has proved to be very useful in our case studies.

Another example of the flexibility allowed by logic variables is in the instantiation of existentials. In traditional first order methods, this is achieved by the technique of introducing a *Skolem function*, e.g., $\forall x.\exists y.P(x,y)$ is transformed into $P(x,f(x))$ where f is a unary function constant with no initial properties. This technique is only of use in refutational proof and the fact that f is a constant is too inflexible in interactive theorem proving. A more “constructive” method is to replace y with the application $Z(x)$ of a function Z to x . Z is a (higher-order) logic variable and is instantiated through unification in the course of doing further proof. At the end of the proof, Z has been filled in with a witness for y .

2.5 Proof management

Just as functions, procedures, and modules provide structuring facilities in programming languages, definitions, lemmas, theories, and parameterized theories are the means for structuring proofs. Beyond that, there is a need for more system support of the user in the course of a proof. In most theorem provers, there is quite primitive support for proof management (two notable exceptions are the KIV and Mural systems). Part of the problem is that system implementors are often more interested in increasing the inference power of their implementations than in providing a nice interface. However, one can largely ignore user interface issues and still provide the following helpful support:

- Backing up out of wrong paths. This is an *undo* facility for proof attempts, and we have found it to be essential.
- Automatic logging of proof transcripts. In many theorem provers, the user has to personally keep track of the proof script. This is quite annoying, especially since “undoing” a proof step may automatically revert the proof state to a previous state, but then the user must manually remove the step from the proof script.
- Proof playback. It is often useful to be able to review a proof, either during the proof, or afterwards. Support for this requires the automatic logging of proof transcripts.
- System support for annotating proofs with commentary either before, during, or after the proof effort. It is common for people to return to a theorem and not be able to understand both the statement and the proof. Commenting a proof offers a solution to this problem, and can also be used to show people how to prove a theorem, or how to use the prover.
- The ability to view the proof at a high level. Often the proof founders in a mass of detail, and the user needs some way to focus on the important aspects. Some

simple facilities are to: only print formulas down to a user-specified level; only show a user-specified subset of the hypotheses; and to only show commentary down to a certain level of the proof.

- Automatic listing of applicable rewrites. Often one knows that a particular subterm needs to be rewritten, but can't remember the name of the theorem to be used as a rewrite rule, or the theory that it is held in. In such cases, a lot of time can be spent searching for the right theorem. A matching procedure could be used to find theorems (perhaps modulo an equational theory) that might rewrite a designated subterm.
- Automatic detection of when a goal is solvable by an automatic method. In a complex theorem proving environment, a user might not know that a particular goal is solvable with an available automatic method, or is simply an instantiation of an existing theorem. In such cases, a *proof assistant* that helpfully pointed out this information would provide valuable support.
- Support for interactive forward proof in the assumptions of a goal. In goal-oriented proof systems, forward proof often becomes a second-class citizen, especially when one wants to do some simple forward inferences involving assumptions of the current goal, *e.g.*, when doing an interactive proof by contradiction. In our view, it should be equally simple to do inference in the conclusion or in the assumptions of a goal. For example, one often wants to eschew the expense of forward chaining in the assumptions in favour of performing a single *modus ponens* operation with two particular assumptions, perhaps to set up a particular rewrite. One approach to this problem might be to allow the attachment of names to hypotheses, although it is our experience that this can cause maintenance problems if automatic procedures are allowed to attach names to hypotheses: when the automatic procedure gets "improved", which is common in the evolution of a theorem prover, it may no longer attach the same names to hypotheses, which inevitably invalidates existing proofs.
- Dependency maintenance for proofs within a theory. A theory ends up being built by a collection of proofs. These proofs have dependencies that should be tracked and, if possible, maintained. If a facility for logging proof transcripts exists, and the dependencies between proofs can be tracked, then it is relatively simple to automatically build the full script that will generate the entire theory. An application of dependency maintenance can be found in the KIV system wherein changes to definitions result in automatic re-execution of proofs so as to maintain consistency with minimal impact to the user[HRS90]. Notice that this is where high-level proof strategies pay off, as they can be reused much more easily. Dependency maintenance allows a high level of interaction with the prover and opens up the possibility of doing *goal directed theory development*, where one works backwards from the theorems that one wants to prove and lets the system keep track of the dependencies that arise.

- Dependency maintenance for theories. A Unix-style *makefile* facility for theories is proving to be very useful in the Isabelle system. In it, whenever a theory file is changed, its descendants automatically become marked so that when the system is re-loaded only the marked theories get re-loaded.
- On-the-fly abbreviation mechanisms are helpful for handling larger proofs where properties can easily be pages long. Some abbreviations, *i.e.*, definitions, are global, but abbreviations can also be local to a proof. For instance, many theorems in mathematics books are stated as “Let $p = \dots, q = \dots$ such that $P(p, q)$ then $Q(p, q)$ holds”. In a theorem prover this could be stated as

$$p = \dots, q = \dots, P(p, q) \vdash? Q(p, q)$$

or, using the power of higher order logic to define an ML-style *let* construct, as

$$\text{let } p = \dots ; q = \dots \text{ in } P(p, q) \supset Q(p, q)$$

It is important, but rare, that the system gives interactive support for the creation and elimination of such abbreviations. As an example of such support, notice that definitions are not only expanded, but also folded. Furthermore, we note that folding such simple definitions as

$$\text{consistent}(S) = \forall i, j. P(S, i) \wedge P(S, j) \supset i = j,$$

go beyond the first-order term rewriting supported in many systems; folding requires higher-order rewriting or at least matching modulo α -conversion.

Of course, many of the the above facilities would become all the more pleasant with window system support, and some others seem only possible with a mouse and bit-mapped screen:

- Direct selection of subterms. Many times, a proof method needs to be directly applied somewhere other than the root of a formula. In such cases, users sometimes have to resort to giving an explicit *occurrence* to target the application. This is very tedious and error-prone: it seems appealing to use the mouse to click on the subterm and have the occurrence automatically computed. It must be noted, however, that this is a very low-level proof technique.
- The theory hierarchy is best visualized as a graph. Nodes of the graph represent theories, and edges represent the *parenthood* relation. Given such a representation, one could click on a theory node and have a window “pop-up” and portray the contents of the theory. More sophisticated capabilities are certainly feasible, such as a comprehensive theory browser/navigator.

2.6 Efficiency

Traditionally, most of the work on efficient, highly-automated theorem proving [WWM⁺90, BM79] has been devoted to first-order languages. Although systems for higher-order logic have a reputation for being slow, there has been remarkable progress in recent years, both in compilers [App92] and implementations [Pfe91, Sli91, Pau90b]. The main observation to be made is that many theorems produced in verifications are primarily first-order. Thus an ideal system should perform in such a way that the higher-order support should not impede efficient handling of first-order terms.

Higher-order patterns [Mil91, Nip93a], which are essentially first-order terms extended by a notion of bound variables, are an interesting compromise between expressive power and efficiency. Patterns can express quantifiers, which need to be explicitly given for natural formalizations. The expressiveness of higher-order patterns is also used extensively in Isabelle to avoid Skolemization and eigenvariable conditions. On the efficiency side of things, patterns have most general unifiers (thus obviating backtracking, a major efficiency problem) and a linear unification algorithm for patterns has recently been presented [Qia93].

A completely different approach to efficiency is *computational reflection*, the iterative strengthening of the prover's implementation, justified by proof [BM81, ACHA90, Sli92]. Roughly, if one can prove that a (representation of a) proof procedure preserves the correctness of the implementation, then the reflection principle would allow that proof procedure to become part of the kernel of the logic implementation. One application of this would be the sound inclusion of efficient versions of complex decision procedures and principles of definition. Reflection is difficult because, among other things, it requires the formalization of the semantics of the implementation programming language. Some researchers have recently claimed that partial evaluation [JSS89] is a lightweight alternative to reflection.

Conclusion

It is interesting to note the many parallels between proof and programming: the tension between expressiveness and efficiency, the necessity of debugging, the importance of mechanized support at all levels of abstraction, and the usefulness of dependency maintenance. In fact, constructive logics [CH85, CAB⁺86] have as an explicit goal the integration of proof and programming.

3 Interface

Since proofs and formalization are done interactively, the interface area requires a great deal more support than in a batch theorem proving environment. In the following, we will deal with the user and programmer interfaces to the system. The user interface is how

most users will deal with the system; the programmer interface is used when customizing the user interface, when writing special-purpose proof support, and also when handling issues relating to interoperability.

3.1 User interface

The formalization of a domain of interest amounts to encoding it in a logic. Once that is accomplished, the sheer size of encoded terms means that they are difficult to read and write; this fact mandates mechanized support for helping the user to mentally map back and forth to the standard way of visualizing objects in the domain. Programs that perform this mapping for the user are known as *parsers* and *prettyprinters*. In the following, we describe the role of such *presentation* support at some different times in the proof development process: syntax definition, formalization, proof, and demonstration.

- During (concrete) syntax definition, the parser(s) and prettyprinter(s) get built. If the user is augmenting the existing interface mechanisms with a new object language, good feedback is necessary to sort out clashes and inconsistencies with existing syntax. Tools developed for the front-end of compilers are generally useful here, such as lexer and parser generators. The interface support defined in this phase is used throughout the other phases. One important feature that the interface should have is *incrementality*: when one starts with a parser and prettyprinter for a logic, then the encoding of various domains of interest should require only incremental changes to the current parser and prettyprinter. Also, it is useful if there is support for multiple syntaxes, *e.g.*, at least plain Ascii and \LaTeX .
- During formalization, the user is entering definitions and needs good feedback with respect to the well-formedness of definitions. As mentioned previously, type inference is quite useful at this stage. One weak point of most ML-style type inference algorithms is in giving good feedback when type inference fails: the algorithm often brings types together from distant parts of a formula; in case they don't unify, it can be difficult to reconstruct where the types came from.
- During proof time, the objects being dealt with are, of course, proofs and relationships between proofs. Usually, the focus is on presenting the state of the current proof, and the suggestions of the “Proof management” section are pertinent.
- During demonstration time, one is trying to show the virtues of a prover, or expost a proof. In these situations, the audience can be highly variable, but won't, in general, want to examine proofs of theorems in full detail, so high level means of presentation need to be found. Any support here seems necessarily *ad hoc*, although the technique of proof documentation discussed in the “Proof management” section can perhaps provide a basis to work from.

It is a real challenge to parse the notations used by mathematicians, since overloaded and ambiguous notation is so often employed. The following are some simple language

features that should be supported in the parser:

- *Overloading* of constants. This is often able to be handled by an extension to the type inference algorithm, although type classes are a more comprehensive solution[WB89].
- The ability to declare the parsing status of constants to be infix (of left or right orientation) with precedence is important.
- *mixfix* parsing declarations: for example, a conditional expression could be mixfix represented as `if __ then __ else __ fi`.
- The ability to declare *binder* constants, such as summation, integration, and differentiation is very useful: in lambda-calculus-based systems, one can use the built-in lambda abstraction to model such binding constructs[Chu40].
- Independence. The primitive operations of the logic should not perform any parsing, for reasons of efficiency and security.
- Parameterization. Another form of independence is to parameterize the implementation of the logic by a parser, thus allowing the user full freedom in the choice of concrete syntax. This is to be used in those situations where an incremental change in the existing parser would not be satisfactory.

There is a range of prettyprinting facilities, ranging from basic but fast programs that insert linebreaks into blocks of text to full scale document preparation systems, such as \TeX . Prettyprinting of large formulas needs to be quick to support interactive proof and hence simple but efficient algorithms based on Oppen's prettyprinter[Opp80] are often used. On the other hand, formulas that need to be read after the proof is finished ought to be typeset, *e.g.*, when being included in a paper. For uniformity's sake, it is useful when the same prettyprinting engine can handle both tasks. Some useful extensions to a basic prettyprinting engine are: support for non fixed-width fonts; and *two dimensional* prettyprinting, so as to portray radicals and the like in Ascii.

User interface factors can influence the operation of proof procedures. For instance, bound variables should be renamed as little as possible, so that the user's original choice of variable names is maintained. (However, a policy of *not* renaming because the underlying implementation uses a deBruijn representation[dB72] is going to be very confusing to the user.) Another problem is that in many systems tactics unnecessarily transform the proof state, requiring extra effort by the user to identify the real changes. For instance, a tactic that blithely reorders the premises may derail certain proof strategies.

Today's bitmapped screens mean that windows, mathematical fonts, colours, and graphics packages can be utilized in presentation. To date, there is no consensus on what a standard logic presentation should be, although there is some software meant to ease the process[Jac93] and some window interfaces to existing theorem provers have been built [TYK92, The93, HRS90]. Visualization of information can ease the proof task, particularly for large proof efforts. Some visualization, *e.g.*, of the theory hierarchy, can be

treated independently from the main proof interface. Keeping such visualizations separate increases modularity and allows individual configuration. The principal challenge is integrating easy to use graphical interfaces with programmable interfaces so that the user can customize the interface.

3.2 Programming interface

Should a user have to, or be allowed to, write programs in order to prove theorems? We will discuss how different proof systems have answered this question. We also discuss interoperability issues, which are assuming greater importance as logic implementations mature.

3.2.1 Access to the metalanguage

An important issue in the design of a theorem prover is how much access to allow to the programming metalanguage. Many LCF-style [GMW79] systems allow the user the full power of ML to build interactive proof procedures: in Isabelle many tactics, *e.g.*, model elimination procedures, are implemented by combining primitive tactics with tacticals implementing search strategies [Pau90a]. Such extensibility comes at a cost: the description of the theorem proving system must include the programming language. Extensible theorem provers are often comfortable for the experienced user, but bewildering to the novice. Other proof systems [CH85, LPT89] deny the user access to the meta-language, allowing only a fixed set of proof commands to be invoked, thus achieving simplicity at the cost of flexibility. In many of our case studies, tacticals were used in the beginning only for building very primitive tactics, and thus might seem superfluous. However, when enough domain specific proof knowledge was developed, special-purpose tactics were usually crafted. We found that, when performing large and often similar proof tasks, it is essential for the user to be able to adapt the system to the domain of formalization. We believe that for any serious production of proofs in a specific domain, access to the metalanguage and the ability to write ones own proof procedures is of fundamental importance.

3.2.2 Interoperability

We can see two scenarios where an interactive theorem prover will need to deal with the “outside world”: first, when another system wants to use the theorem prover as a subcomponent; second, when the theorem prover wants to get “answers” to help it with a proof.

- There will be systems that want to incorporate a theorem prover as a subcomponent. To provide for that eventuality, the theorem prover must encapsulate its data structures and algorithms, and it must provide an explicit interface for clients to use. This includes types, input and output mechanisms for formulae, proof support,

and, importantly, the errors that can be raised in the proof module and percolate to the client.

- Often there will be an external source of information (call it an *oracle*) that can solve a problem raised inside a theorem prover. How should this information be accessed and treated? An interesting example of this is interacting with a computer algebra system [HT93]. At times, the theorem prover will want to use a result from an oracle as a theorem, but simple acceptance of such results means that the correctness of the theorem prover is dependent on the correctness of the oracle (which we hold to be an unsatisfactory state of affairs). We know of two ways to handle this:
 - Check that the result from the oracle is in fact a theorem, by proving it inside the logic (in many cases this can be automated[HT93, Sli93]).
 - Tag the result and propagate the tag to every theorem derived from the result. In case the oracle is flawed (as is often the case), a false theorem might get proved in the proof system, but the tag would show which oracle(s) were at fault. This idea is due to Mike Gordon, who found a clever way to use the devices of the logic to propagate tags[Gor93].

4 Other considerations

There are a couple of other topics that resist the above classification into formalization, proof, and interface:

- Correctness. Since theorem provers have been proposed as a method for helping produce safety-critical hardware and software, it is of great importance that a theorem prover itself be correct. Difficult algorithms commonly used in such implementations, such as beta-conversion and higher-order unification, mean that this is not an easy task. To be done properly, this approach requires that the semantics of the implementing programming language be formalized first. This can be a problem in that the “right” semantics for such languages as ML has not yet been settled. As an intermediate step, several researchers have specified the logic implementation abstractly [Art90, BD93]. Another approach that would sidestep the need for all this formalization would be to require the theorem prover to emit explicit proofs, and have a separate small and easily verified *proof checker* program that went over the proof and simply decided whether it was a valid proof or not[Won93]. A problem here is the size of proofs and whether they can be feasibly checked.
- Documentation. Online help is standard support for any interactive program, however many theorem proving implementations lack this vital facility. This is mainly due to the fact that most systems are still research prototypes. Extensible theorem provers suffer in the provision of online help, because users are often slack with regard to documenting their extensions.

5 Conclusion

We have neglected many worthwhile topics in trying to give a relatively compact overview of tools for interactive verification systems. In spite of that, we have identified the following important services:

- powerful definition facilities are a fundamental requirement for the secure development of theories;
- polymorphism and abstract theories offer reuse in the logic;
- automatic type inference offers a quick consistency check for term construction and frees the user from having to declare the types of variables;
- flexible and programmable proof support with incremental proof procedures allows the user to control how the proof unfolds and how it is presented;
- a high level of proof automation is required to let the user confront the real problems in large verifications. To this end, relatively efficient implementations of decision procedures such as that for unquantified Presburger arithmetic are quite useful;
- strong proof management facilities are essential to enable the user to keep a firm grip on the development of the theory; and
- presentation facilities such as overloading, incremental parsing, and prettyprinting are essential when trying to satisfy the notational needs of mathematicians and computer scientists at different stages of their work.

Some of these facilities we have already in our systems; others, like automation and proof management are moving targets where there is always more that can be done. We have not tried to list facilities that a “dream” theorem prover would supply since we are a long way from being able to achieve that. The facilities we have listed throughout this document are achievable in the short term, and if properly integrated they will give us proof environments that are vastly better than those currently available.

Acknowledgements

We thank John Harrison and Richard Boulton for their helpful comments on this paper.

References

- [ACHA90] S.F. Allen, R. Constable, D. Howe, and W. Aitken. The semantics of reflected proof. In *Fifth annual IEEE symposium on Logic in Computer Science*, pages 95–107, Philadelphia, USA, June 1990.

- [AL93] Mark Aagaard and Mirian Leeser. A theorem proving based methodology for software verification. Technical report, Cornell University, May 1993. submitted to IEEE Transactions on Software Engineering.
- [AMC84] Peter Andrews, Dale Miller, and Eve Longini Cohen. Automating higher order logic. In Woody Bledsoe and Donald Loveland, editors, *Automated Theorem Proving: After 25 Years*, volume 29 of *Contemporary Mathematics Series*, pages 169–192. American Mathematical Society, 1984.
- [App92] Andrew Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
- [Art90] R.D. Arthan. A formal specification of HOL. Technical Report DS/FMU/IED/SPC001, ICL Defence Systems, April 1990.
- [BD93] R.S. Boyer and G. Dowek. Towards checking proof checkers. In Herman Geuvers, editor, *Informal Proceedings of the Workshop on Types for Proofs and Programs*, pages 51–70, Nijmegen, Netherlands, May 1993.
- [BDD⁺92] Manfred Broy, Franck Diederichs, Claus Dendorfer, Max Fuchs, Thomas Gritzner, and Rainer Weber. The design of distributed systems — an introduction to Focus. Technical Report I9202, Institut für Informatik, Technische Universität München, January 1992.
- [BF93] P. Baumgartner and U. Furbach. Model elimination without contrapositives and its application to pttp. Technical Report 12/93, Universität Koblenz, 1993. (submitted to CADE 12).
- [BG93] Leo Bachmair and Harald Ganzinger. Rewrite techniques for transitive relations. Technical Report MPI-I-93-249, Max-Planck-Institut für Informatik, Saarbrücken, Germany, November 1993.
- [BGM88] P. G. Bosco, E. Giovanetti, and C. Moiso. Narrowing vs. SLD-resolution. *TCS*, 59:3–23, 1988.
- [BJ89] George Boolos and Richard Jeffrey. *Computability and Logic*. Cambridge University Press, 3 edition, 1989.
- [BM79] Robert S. Boyer and J. Strother Moore. *A Computational Logic*. Academic Press, 1979.
- [BM81] R.S Boyer and J.S. Moore. Meta-functions: proving them correct and using them as new proof procedures. In R.S. Boyer and J.S. Moore, editors, *The Correctness Problem in Computer Science*, pages 103–184, Liege, Belgium, 1981.
- [BM90] R.S. Boyer and J S. Moore. A theorem prover for a computational logic. In Mark Stickel, editor, *Proceedings of the Tenth International Conference on Automated Deduction, LNAI 449*, pages 1–15, Kaiserslautern, 1990.

- [Bro92] Manfred Broy. Experiences with software specification and verification using LP, the Larch proof assistant. Technical Report 93, DIGITAL Systems Research Center, November 1992.
- [Bry86] Randal Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
- [BW88] Richard Bird and Philip Wadler. *Introduction to functional programming*. Prentice Hall, 1988.
- [CAB⁺86] Robert Constable, S. Allen, H. Bromly, W. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P. Panangaden, J. Sasaki, and S. Smith. *Implementing Mathematics With the Nuprl Proof Development System*. Prentice-Hall, New Jersey, 1986.
- [CES86] E. Clarke, E. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.
- [CH85] Thierry Coquand and Gerard Huet. Constructions: A higher order proof system for mechanizing mathematics. In *Eurocal '85, Proceedings of the European Conference on Computer Algebra (LNCS 203)*, volume 1, pages 136–150, Linz, Austria, April 1985. Springer-Verlag.
- [Chu40] Alonzo Church. A formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.
- [CJ90] J.H. Cheng and C.B. Jones. On the usability of logics which handle partial functions. In C. Morgan and J. Woodcock, editors, *Third Refinement Workshop*, Springer Verlag Workshops in Computing, Hursley Park, Great Britain, 1990.
- [CPS89] Rance Cleaveland, Joakim Parrow, and Bernhard Steffen. The concurrency workbench. In J.Sifakis, editor, *Automatic Verification Methods for Finite State Systems (LNCS 407)*, pages 24–37, 1989.
- [CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.
- [dB72] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indag. Math.*, (34):381–392, 1972.
- [Der87] Nachum Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987.
- [Dow90] Gilles Dowek. A proof synthesis algorithm for a mathematical vernacular. In Gerard Huet and Gordon Plotkin, editors, *Informal Proceedings of the First Workshop on Logical Frameworks*, pages 183–213, Antibes, France, May 1990. electronically distributed.

- [Fay79] Michael Fay. First-order unification in an equational theory. In William H. Joyner, editor, *Proceedings of the Fourth Workshop On Automated Deduction. Austin, Texas, 1979*. University Press, 1979.
- [FFMH92] Mick Francis, Simon Finn, Ellie Mayger, and Roger Hughes. Reference manual for the lambda system. Technical Report 4.2.1, Abstract Hardware Limited, 1992.
- [FGT90] William Farmer, Joshua Guttman, and Javier Thayer. Imps: an interactive mathematical proof system. In Mark Stickel, editor, *Tenth International Conference on Automated Deduction (CADE)*, pages 653–654, Kaiserslautern, 1990.
- [GG89] Stephen J. Garland and John V. Guttag. An overview of LP, the Larch prover. In *Proceeding of RTA-89*, pages 137–151, Dijon France, April 1989. LNCS 455.
- [GM93] Mike Gordon and Tom Melham. *Introduction to HOL, a theorem proving environment for higher order logic*. Cambridge University Press, 1993.
- [GMW79] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
- [Gor85] Mike Gordon. Why Higher-Order Logic is a good formalism for specifying and verifying hardware. Technical Report 77, Computer Laboratory, The University of Cambridge, 1985.
- [Gor93] Michael Gordon. Note to info-hol mailing list, archive number 0914. Tagging theorems for proof acceleration, January 1993.
- [Gri88] D. Grigor’ev. Complexity of deciding Tarski Algebra. *Journal of Symbolic Computation*, 5:65–108, 1988.
- [GV88] D. Grigor’ev and N Vorobjov. Solving systems of polynomial inequalities in subexponential time. *Journal of Symbolic Computation*, 5:37–64, 1988.
- [Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.
- [HRS90] M. Heisel, W. Reif, and W. Stephan. Tactical theorem proving in program verification. In Mark Stickel, editor, *Proceedings of the Tenth International Conference on Automated Deduction, LNAI 449*, pages 1–15, Kaiserslautern, 1990.
- [HRS93] J. Heintz, M-F. Roy, and P. Solerno. On the theoretical and practical complexity of the existential theory of reals. *The Computer Journal*, 36(5):427–431, 1993. Special issue on Computational Quantifier Elimination.

- [HT93] John Harrison and Laurent Thery. Reasoning about the reals: The marriage of HOL and Maple. In *LPAR 93 (LNAI 698)*, St. Petersburg, Russia, 1993.
- [Hue81] Gerard Huet. A complete proof of correctness of the Knuth-Bendix completion algorithm. *Journal of Computer and System Sciences*, 21:11–21, 1981.
- [Jac93] I. Jacobs. The Centaur 1.3 manual. Technical report, INRIA-Sophia-Antipolis, January 1993.
- [JK93] J. Joyce and K. Seger, editors. *International Workshop on Higher Order Logic Theorem Proving and its Applications*, Vancouver, Canada, August 1993. Springer Verlag.
- [JMSY90] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP(\mathcal{R}) language and system. Research Report RC 16292 (#72336), IBM T.J. Watson Research Center, November 1990.
- [JSS89] N.D. Jones, P. Sestoft, and H. Sondergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.
- [KB70] Donald Knuth and Peter Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*. Pergamon Press, Oxford, 1970.
- [KN92] Deepak Kapur and Paliath Narendran. Double-exponential complexity of computing a complete set of AC-unifiers. In *Proceeding of the Seventh Annual IEEE Symposium on Logic in Computer Science*, Santa Cruz, California, June 1992. Computer Society Press.
- [Kup93] Jan Kuper. An axiomatic theory for partial functions. *Information and Computation*, 107(1), November 1993.
- [Lam93a] Leslie Lamport. How to write a long formula. Technical report, DEC Systems Research Center, November 1993. unpublished.
- [Lam93b] Leslie Lamport. How to write a proof. Technical Report 94, DEC Systems Research Center, Palo Alto, California, February 1993.
- [LPT89] Z. Luo, R. Pollack, and P. Taylor. How to use LEGO (a preliminary user’s manual). Technical Report LFCS-TN-27, Department of Computer Science, Edinburgh University, 1989.
- [LT89] N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, September 1989.
- [McA89] David A. McAllester. *ONTIC: A Knowledge Representation System for Mathematics*. MIT Press, 1989.

- [Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
- [Mil84] Dale Miller. Expansion tree proofs and their conversion to natural deduction proofs. In Robert Shostak, editor, *Seventh International Conference on Automated Deduction (LNCS 170)*, pages 375–393, Napa, California, USA, May 1984.
- [Mil85] Robin Milner. The use of machines to assist in rigorous proof. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, International Series in Computer Science, pages 77–87. Prentice-Hall, 1985.
- [Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming (LNCS 475)*, pages 253–281. Springer-Verlag, 1991.
- [MN90] Ursula Martin and Tobias Nipkow. Ordered rewriting and confluence. In M.E. Stickel, editor, *Proc. 10th Int. Conf. Automated Deduction*, pages 366–380. LNCS 449, 1990.
- [Nip93a] Tobias Nipkow. Functional unification of higher order patterns. In *Proceedings of the Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 64–74, Montreal, Canada, June 1993. IEEE Computer Society Press.
- [Nip93b] Tobias Nipkow. Order-sorted polymorphism in Isabelle. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 164–188. Cambridge University Press, 1993.
- [NO79] Greg Nelson and Derek Oppen. Simplification by cooperating decision procedures. *Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.
- [NO80] Greg Nelson and Derek Oppen. Fast decision procedures based on congruence closure. *Journal of the Association for Computing Machinery*, 27(2):356–364, April 1980.
- [Opp80] Derek Oppen. Prettyprinting. *Transactions on Programming Languages and Systems*, 2(4):465–483, October 1980.
- [ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction*, LNAI 607, pages 748–752, Saratoga Springs, New York, USA, June 15–18, 1992. Springer-Verlag.
- [ORSvH93] S. Owre, J. Rushby, N. Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: some lessons learned. In J. Woodcock

and P. Larsen, editors, *FME'93: Industrial Strength Formal Methods (LNCS 670)*, pages 482–500, Odense Denmark, 1993.

- [Pau83] Lawrence Paulson. A higher order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.
- [Pau87] Lawrence Paulson. *Logic and Computation: Interactive Proof With Cambridge LCF*. Cambridge University Press, 1987.
- [Pau90a] Lawrence Paulson. Designing a theorem prover. Technical Report 192, University of Cambridge, 1990.
- [Pau90b] Lawrence Paulson. Isabelle: The next 700 theorem provers. In P.G. Oddifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.
- [Pfe91] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon D. Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991.
- [Pre93a] Christian Prehofer. Decidable higher-order unification problems. Technical report, TU München, Institut für Informatik, 1993. Submitted.
- [Pre93b] Christian Prehofer. Solving higher-order equations. Technical report, TU München, Institut für Informatik, 1993. Submitted.
- [PS81] Gary Peterson and Mark Stickel. Complete sets of reductions for some equational theories. *Journal of the Association for Computing Machinery*, 28:233–264, 1981.
- [Qia93] Zhenyu Qian. Linear unification of higher-order patterns. In J.-P. Jouanaud, editor, *Proceedings of 1993 Colloquium on Trees in Algebra and Programming*. LNCS, 1993. to appear.
- [Reg94] Franz Regensburger. *HOLCF: Eine konservative Einbettung von LCF in HOL*. PhD thesis, Institut für Informatik, Technische Universität München, 1994.
- [Sco93] Dana Scott. A type theoretic alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121(1-2):411–440, December 1993.
- [Sho79] Robert Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the Association for Computing Machinery*, 26(2):351–360, April 1979.
- [Sho84] Robert Shostak. Deciding combinations of theories. *Journal of the Association for Computing Machinery*, 31(1):1–12, January 1984.

- [Sie84] Jorg Siekmann. Universal unification. In Robert Shostak, editor, *Proceedings of the Seventh International Conference on Automated Deduction (LNCS 170)*, pages 1–42, Napa, California, USA, May 1984. Springer-Verlag.
- [Sli91] Konrad Slind. An implementation of higher order logic. Technical Report 91-419-03, University of Calgary Computer Science Department, 1991.
- [Sli92] Konrad Slind. Adding new rules to an LCF-style logic implementation: Preliminary report. In L. Claesen and M. Gordon, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, Leuven, Belgium, September 1992. IFIP TC10/WG10.2, Elsevier Science Publishers.
- [Sli93] Konrad Slind. AC unification in hol90. In J. Joyce and K. Seger [JK93].
- [The93] L. Thery. A proof development system for HOL. In J. Joyce and K. Seger [JK93].
- [TYK92] L. Thery, Y. Bertot, and G. Kahn. Real theorem provers deserve real user-interfaces. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments (Software Engineering Notes)*, volume 17, pages 365–380, Tyson’s Corner, Virginia USA, 1992. ACM Press.
- [WB89] P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *Proc. 16th ACM Symp. Principles of Programming Languages*, pages 60–76, 1989.
- [Wir90] M. Wirsing. Algebraic specification. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume 2, pages 675–788. North Holland, 1990.
- [Won93] Wai Wong. Recording HOL proofs. University of Cambridge Computer Laboratory Technical Report 306, University of Cambridge Computer Laboratory, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, 1993.
- [WWM⁺90] L. Wos, S. Winker, W. McCune, R. Overbeek, E. Lusk, R. Steven, and R. Butler. Automated reasoning contributes to mathematics and logic. In Mark Stickel, editor, *Proceedings of the Tenth International Conference on Automated Deduction, LNAI 449*, pages 485–499, Kaiserslautern, 1990.