Using Decision Procedures With a Higher-Order Logic*

Natarajan Shankar

Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
{shankar, owre}@csl.sri.com
URL: http://www.csl.sri.com/˜shankar/
Phone: +1 (650) 859-5272 Fax: +1 (650) 859-2844

Abstract. In automated reasoning, there is a perceived trade-off between expressiveness and automation. Higher-order logic is typically viewed as expressive but resistant to automation, in contrast with first-order logic and its fragments. We argue that higher-order logic and its variants actually achieve a happy medium between expressiveness and automation, particularly when used as a front-end to a wide range of decision procedures and deductive procedures. We illustrate the discussion with examples from PVS, but some of the observations apply to other variants of higher-order logic as well.

It does not matter if a cat is black or white so long as it catches mice.

Deng Xiaoping

Expressiveness and automation are the two basic concerns for formalized mathematics. In theoretical terms, expressiveness is characterized by the range of concepts that can be captured within a given logic. For practical application of logic, the directness with which concepts can be captured and manipulated in the logic is also important. The theoretical idea of automation for a given logic is usually in terms of the degree of solvability or unsolvability of its decision problem. We argue that such characterizations are theoretically interesting but irrelevant in practice. A more important quality for a logic is the degree to which useful fragments of the logic can be supported by means of automation. We have used the higher-order logic of PVS (SRI's Prototype Verification System) [ORS92,ORSvH95] as an interface to various automated procedures. We assess the effectiveness of the PVS logic, and higher-order logic in general, as a logical front end.

* This work was funded by NSF Grant CCR-0082560, DARPA/AFRL Contract F33615-00-C-3043, and NASA Contract NAS1-00079.

Logics based on higher-order types have a long and distinguished history in the foundations of mathematics. Paradoxes such as that of Russell (the set of all those sets that do not contain themselves as elements) led to various restrictions on hitherto unquestioned logical principles such as unrestricted comprehension. Whitehead and Russell's system Principia Mathematica (PM) [WR27] introduced a type system consisting of the types of individuals, sets of individuals, sets of sets of individuals, and so on. Overlaid on this type hierarchy was a notion of order so that a term of order k could only contain quantification over variables of lower order to ensure predicativity. However, as Chwistek (see Hatcher [Hat68]) showed, the axiom of reducibility (comprehension) renders the system equivalent to a simple type theory with impredicative comprehension. Ramsey [Ram90] showed that the basic type system was sufficient to avoid the logical paradoxes (such as Russell's). Predicativity did not actually address the epistemological paradoxes (such as the Liar's paradox) since these had to do with the interpretation of concepts expressed in the logic and not with the means of definition. Church [Chu40] presented a system based on a simple type theory with types for individuals and propositions, and a type $[A \rightarrow B]$ for any types A and B. Church's system is based on the simply typed lambda-calculus with Church numerals, an axiom of infinity, and operators for universal quantification and choice. Church's system is equivalent in power to Zermelo's set theory. Most modern higher-order logics are based on Church's system.[1]

Many automated reasoning systems have been based on higher-order logics. Perhaps the earliest of these is de Bruijn's Automath proof checker [dB80] which employs a dependently typed metalogic to represent judgements and derivations. The TPS system [AINP88] is one of the earliest automated theorem provers based on a higher-order logic. TPS has been used to automatically find succinct higher-order proofs for several interesting theorems. Many of the systems developed during the 1980s such as HOL [GM93], Veritas [HD92], and EHDM [vHCL$^+$88,RvHO91] are based on higher-order logic. The Nuprl system [CAB$^+$86] is based on a constructive system of higher types and universes inspired by the type theories of Martin-Löf. The Coq system [CH85,DFH$^+$91] also features a similar constructive type theory with impredicative type quantification and abstraction. Isabelle [Pau94] is a metalogical framework that uses higher-order Horn clauses to represent inference rules. λ-Prolog [NM90] is a similar metalogical framework based on the hereditary-Harrop fragment of higher-order logic.

There is a general impression that higher-order logics are not easily automated. We get a different picture if we view higher-order logic as a framework for embedding small and useful sublogics that are more readily amenable to automation [Gor88]. The convenience of higher-order logic is that other formalisms including program logics, modal and temporal logics, can be embedded directly

[1] Excellent surveys of the logical aspects of higher-order logic have been given by Feferman [Fef78], Andrews [And86], van Bentham and Doets [vBD83], and Leivant [Lei94].

and elegantly within it. Given such embeddings, it is easy to recognize those fragments that are well-supported by automation. The decision as to whether a problem is in a decidable fragment can itself be undecidable. By allowing other logics to be embedded within it, higher-order logic makes it easy to recognize when an assertion falls within a decidable fragment. The dialogue between the proof designer and the theorem prover is guided by the goal of reducing a given problem into a form that has been effectively automated. The sharing of results between different automated deductive procedures is straightforward because the higher-order logic serves as a common semantic medium. The term *glue logic* has been applied to this function, but higher-order logic is not merely an interlingua. It supports both the direct expression and effective manipulation of formalized mathematics.

Boyer and Moore [BM86] have already argued that the integration of decision procedures into an automatic theorem prover can be a delicate matter. This is even more acutely the case with an interactive prover. Decision procedures must ensure that the interaction is conducted at a reasonably high-level by eliminating trivial inferences. At the same time, the internal details of the decision procedures must, to the extent possible, be concealed from the human user. On the other hand, decision procedures can be used with greater resilience in an interactive context through the use of contextual knowledge and the "productive use of failure" [Bun83].

Highly automated deduction procedures are crucial for achieving useful results within specific domains. The primary challenge for interactive theorem proving is that of integrating these procedures within a unifying logic so that they can be combined into a proof development in a coherent manner. In contrast, the Automath [dB80] and LCF [GMW79] systems emphasize formality, foundational reductions, and fully expansive formal proof construction over automation. In the LCF approach, ML is used as a metalanguage in which theorems are an inductively defined datatype, *tactics* are defined as a way of introducing derived inference rules in a sound manner, and *tacticals* are used for combining tactics.

Soundness is an obviously critical property of a deductive procedure, but *manifest soundness* as ensured by *fully expansive* proofs must be balanced against the efficiency of the procedure, its transparency, usability, and interoperability with other procedures. Fully expansive proof construction [Bou93,Har96] is a powerful organizing principle and an effective technique for achieving soundness, but at the algorithmic level, it requires a serious investment of time and effort that can be justified only after a great deal of experimentation. Eventually, it will be both prudent and feasible to verify the soundness of these deductive procedures (as demonstrated by Théry's verification [Thé98] of Buchberger's algorithm), but expanded proofs might still be useful for applications such as proof-carrying code [NL00].

The initial goal in the development of PVS was not dissimilar to that of the LCF family of systems with a fixed set of inference rules and a tactic-like mechanism for combining useful patterns of inference. More recently, we have come to view PVS as a semantic framework for integrating powerful deductive procedures. This viewpoint is similar to that of the PROSPER project [DCN$^+$00] though PVS employs a tighter and more fine-grained coupling of deductive procedures. Other ways of combining PVS with other existing automation have been studied by Martin and her colleagues [DKML99] for adding soundness checks to computer algebra systems, Buth [But97] for discharging proof obligations generated by PAMELA from VDM-style specifications, and the LOOP system of Jacobs and his colleagues [JvdBH$^+$98] for verifying object-oriented systems described in Java, among others.

We outline the PVS logic (Section 1) and present a few examples of deductive procedures that have been used to automate proof development in its higher-order logic. The deductive procedures include extended typechecking (Section 2), evaluation (Section 3), ground decision procedures for equality (Section 4), and symbolic model checking (Section 5). Most of these procedures can be employed within modest extensions to first-order logic, but the use of higher-order logic extends the range of applicability without compromising the automation. Many of the methods have been described in detail elsewhere and not all of these are original with PVS. The PVS system is a work in progress toward a vision of interactive proof checking based on exploiting the synergy between expressiveness and automation. The present paper is a progress report on some of the experiments on this front and an attempt to connect the dots.

1 The PVS Higher-Order Logic

The core of the PVS higher-order logic is described in the formal semantics of PVS [OS97]. The types consist of the base types `bool` and `real`. Compound types are constructed from these as tuples $[T_1, \ldots, T_n]$, records $[\#l_1 : T_1, \ldots, l_n : T_n\#]$, or function types $[S{\rightarrow}T]$. Predicates over a type T are of type $[T{\rightarrow}\texttt{bool}]$. Predicate subtypes are a distinctive feature of the PVS higher-order logic. Given a predicate p over T, $\{x : T \mid p(x)\}$ is a predicate subtype of T consisting of those elements of T satisfying p. Dependent versions of tuple, record, and function types can be constructed by introducing dependencies between different components of a type through predicates. Let $T(x_1, \ldots, x_n)$ range over a type expression that contains free variables from the set $\{x_1, \ldots, x_n\}$. A dependent tuple has the form $[x_1 : T_1, x_2 : T_2(x_1), \ldots, T_n(x_1, \ldots, x_{n-1})]$, a dependent record has the form $[\#x_1 : T_1, x_2 : T_2(x_1), \ldots, x_n : T_n(x_1, \ldots, x_{n-1})\#]$, and a dependent function type has the form $[x : S{\rightarrow}T(x)]$.

Predicate subtyping can be used to type partial operations but it can also be used to declare more refined type information than is possible with simple types [ROS98]. As examples, the division operation is declared to range over non-zero denominators, and the square-root operation is declared to take a non-negative argument. The range of an operation can also be constrained by subtyping so that the absolute-value function can be asserted to return a non-negative result. At the first-order level, types such as the natural numbers, even numbers, prime numbers, subranges, nonempty lists, ordered binary trees, and balanced binary trees, can be defined through predicate subtyping. An empty subtype of any given type can be easily defined. At higher types, predicate subtyping can be used to define types such as injective, surjective, bijective, and order-preserving maps, and monotone predicate transformers. Arrays can be typed as functions with subranges as domains. Dependent typing at higher types admits, for example, the definition of a finite sequence (of arbitrary length) as a pair consisting of a length and an array of the given length. Length or size preserving maps can be captured by dependent types. A sorting operation can be typed to return an ordered permutation of its argument. A choice operation that picks an arbitrary element from a nonempty set (predicate) can be captured by dependent typing.

Predicate subtypes and dependent types are conservative extensions to the simply typed higher-order logic. Any concept that can be defined with the use of subtypes and dependent types, can be captured within the simple type system. As examples, division by zero can be defined to return a chosen arbitrary value, and finite sequences can be defined from infinite sequences by ignoring the irrelevant values. However, this embedding has the disadvantage that the equality operation on finite sequences has to be defined to ignore these irrelevant elements or they have to be coerced to a fixed element. Either option adds some overhead to the formal manipulations involving such types.

In addition to variables x, constants c, applications $f(a)$, and abstractions $\lambda(x:T):a$, PVS expressions include conditionals $\texttt{IF}\ a_1\ \texttt{THEN}\ a_2\ \texttt{ELSE}\ a_3\texttt{ENDIF}$, and updates $e[a := v]$ where e is of tuple, record, or function type and a is respectively, a tuple component number, a record field name, or a function argument.[2] The update construct is an interesting extension. Although, for a given type, the update operation can be defined by means of copying, there are some important differences. In executing higher-order specifications as programs, it is possible to identify update operations that can be executed by a destructive or in-place udpate. Such destructive updates yield substantial efficiencies over copying. The update operation has the property of being polymorphic with respect to structural subtyping.[3]

2 Typechecking in PVS

Typechecking of PVS terms is undecidable in general because it can yield arbitrary formulas as proof obligations. For example, the singleton subtype of the

[2] The actual update construct also admits simultaneous and nested updates.

[3] Structural subtyping is different from predicate subtyping. For example, a record type r that has fields l_1, l_2, and l_3, of types T_1, T_2, and T_3, is a structure subtype of a record s with field l_1 and l_2 with types T_1 and T_2. Updates are structural subtype polymorphic in the sense that any update operation that maps s to s is also a valid update from r to r. Structural subtyping is not yet part of the PVS logic but it nevertheless illustrates the nontriviality of the update operation.

booleans consisting of the element TRUE is a valid type for any valid sentence in the logic. Other judgements such as equalities between types, the subtyping relation between types, and the compatibility between types are also similarly undecidable.

Typechecking of a PVS expression a with respect to a context Γ which is a list of declarations for type names, constants, and variable names, is given by means of an operation $\tau(\Gamma)(a)$ that either signals an error, or computes the canonical type of a with respect to Γ and a collection of proof obligations ψ. Typechecking is usually specified by type inference rules, but we have opted for a functional presentation because it gives a deterministic algorithm that constructs a canonical type for a given term. Proof obligations are generated with respect to this canonical type. The semantic mapping for a well-formed term is also given by a recursion similar to that of τ, and the soundness proofs follow the same structure. Type correctness of a with respect to Γ holds when all the proof obligations have been discharged.

The typechecking of application expressions is worth examining in some detail since it captures the key ideas. The context Γ includes formulas corresponding to any governing conditions, in addition to declarations. The corresponding case in the definition of τ is given below.

$$\begin{array}{ll} \tau(\Gamma)(f\ a) = B', \text{ where} & \mu_0(\tau(\Gamma)(f)) = [x : A \rightarrow B], \\ & \tau(\Gamma)(a) = A', \\ & (A \overset{a}{\sim} A')_\Gamma, \\ & B' \text{ is } B[a/x], \\ & \vdash_\Gamma \pi(A)(a) \end{array}$$

The body of the definition for $\tau(\Gamma)(f\ a)$ first computes the canonical type $\tau(\Gamma)(f)$ for f and the canonical type $\tau(\Gamma)(a)$ as A'. The direct maximal supertype of $\tau(\Gamma)(f)$ must be a possibly dependent type of the form $[x : A \rightarrow B]$ for some types A and B, or typechecking fails. The expected type A and actual type A' must be compatible, i.e., $(A \overset{a}{\sim} A')_\Gamma$, and this compatibility test generates proof obligations that must be discharged. The types int of integers, and nat of natural numbers, are compatible since they have a common supertype. The types $[\texttt{nznat} \rightarrow \texttt{int}]$ and $[\texttt{posnat} \rightarrow \texttt{nat}]$, where nznat is $\{x : \texttt{nat} \mid x \neq 0\}$ and posnat is $\{x : \texttt{nat} \mid x > 0\}$, are also compatible since the domain types are equal (modulo the verification of the proof obligation $\forall (x : \texttt{nat}) : x \neq 0 \iff x > 0$), and the range types are compatible. Finally, any proof obligations corresponding to constraints on a with respect to the expected type A, namely $\pi(A)(a)$ must also be discharged.

Typical proof obligations involve little more than subranges and interval constraints. For example, arrays in PVS are just functions on subrange types and array bounds checking ensures that for any array access $a(i)$, i occurs within the subrange corresponding to the index type of a. Proof obligations corresponding to subtype constraints in the range are generated, for example, to demonstrate that the absolution-value operation returns a non-negative result. Range subtype constraints are interesting because they can be used to verify properties that might otherwise require inductive reasoning. For example, a function which halves an even integer can be defined as

$$
\begin{array}{l}
\mathit{half}(x : \texttt{even}) : \{y : \texttt{int} \mid 2 * y = x\} \\
= \begin{array}[t]{ll}
\texttt{IF} & x = 0 \\
\texttt{THEN} & 0 \\
\texttt{ELSIF} & x > 0 \\
\texttt{THEN} & \mathit{half}(x - 2) + 1 \\
\texttt{ELSE} & \mathit{half}(x + 2) - 1 \\
\texttt{ENDIF}.
\end{array}
\end{array}
$$

The type `even` is defined as $\{x : \texttt{int} \mid \mathit{even}?(x)\}$. The range type captures the constraint that $2 * \mathit{half}(x) = x$. The subtype proof obligations generated are

1. $\forall(x : \texttt{even}) : x = 0 \supset 2 * 0 = 0$. The return value 0 when $x = 0$ must satisfy the subtype $\{y : \texttt{int} \mid 2 * y = x\}$.
2. $\forall(x : \texttt{even}) : x \neq 0 \wedge x > 0 \supset \mathit{even}?(x - 2)$. The argument $x - 2$ in the recursive call $\mathit{half}(x - 2)$ must satisfy the constraint given by the subtype `even`.
3. $\forall(x : \texttt{even}, z : \{y : \texttt{int} \mid 2 * y = x - 2\}) : x \neq 0 \wedge x > 0 \supset 2 * (z + 1) = x$. The return value $\mathit{half}(x - 2) + 1)$ satisfies the constraint given by the range type $\{y : \texttt{int} \mid 2 * y = x\}$.
4. $\forall(x : \texttt{even}) : x \neq 0 \wedge \neg x > 0 \supset \mathit{even}?(x + 2)$. The recursive call argument $x + 2$ satisfies the constraint given by `even`.
5. $\forall(x : \texttt{even}, z : \{y : \texttt{int} \mid 2 * y = x + 2\}) : x \neq 0 \wedge \neg x > 0 \supset 2 * (z - 1) = x$. The return value $\mathit{half}(x + 2) - 1)$ satisfies the constraint given by the range type $\{y : \texttt{int} \mid 2 * y = x\}$.

Proof obligations are also generated to ensure the termination of recursive definitions. For example, the above definition of *half* can be shown to be well-formed with respect to termination by supplying the termination measure $\lambda(x : \texttt{even}) : \mathit{abs}(x)$ and the well-founded ordering relation $<$ on natural numbers. The termination proof obligations generated are

1. $\forall(x : \texttt{even}) : x \neq 0 \wedge x > 0 \supset \mathit{abs}(x - 2) < \mathit{abs}(x)$.
2. $\forall(x : \texttt{even}) : x \neq 0 \wedge \neg x > 0 \supset \mathit{abs}(x + 2) < \mathit{abs}(x)$.

All these proof obligations can be trivially discharged using decision procedures such as those outlined in Section 4. Conversely, the subtype constraints are exploited by the ground decision procedures. Since it is possible to build in arbitrarily complex subtype constraints, some proof obligations might require interactive proof. The handling of predicate subtypes requires a fair bit of infrastructure. Subtyping *judgements* can be used to cache subtype information. One kind of subtyping judgement merely introduces more refined typing on an existing constant. For example, the numeral constant 2 can be introduced as an natural number, but judgements can be introduced to assert that 2 is also an even number and a prime number. The subtyping judgements generate proof obligations that must be discharged for type correctness. Subtyping judgements are invoked by the typechecker for annotating terms and propagating type information that is more refined than the canonical type so that the corresponding type correctness proof obligations are not generated. For the absolute-value operation *abs* of type [`real`→`real`], a judgement can be used to assert that $abs(x)$ has type `int` if x has type `int`. This way, when the term $abs(a)$ is typechecked where a has type `int`, the typechecker deduces that $abs(a)$ also has type `int`.

Another form of judgement is used to record a subtyping relation between two types. For example, a judgement can be used to note that a subrange from a positive natural number up to another natural number is a subtype of the positive natural numbers. Judgements are similar to the intersection type discipline [CDC80] for Curry-style type inference systems. PVS uses a Church-style type discipline and judgements are used for type propagation rather than inference.

3 Executing PVS Specifications

Evaluation is the simplest of all decision procedures. It has many uses within a proof checker. First, specifications can be validated by executing them on known test suites. This is a useful way of ensuring that the specifications capture the specifier's intent. Second, evaluation can be used to obtain executable code from a specification so that the program that is executed is essentially identical to the program that has been verified. Third, evaluation can be used to efficiently simplify executable expressions. Fourth, evaluation can be used to add a reflective capability so that metatheorems involving deductive procedures for a logic can be verified in the logic itself, and the resulting procedures can be efficiently executed.[4] An evaluation capability was added to PVS in response to a practical need identified by researchers at Rockwell Collins [HWG98] for validating hardware descriptions used in processor verification.

There are several options in designing an execution capability for PVS. One option is to build an interpreter for the language, but its efficiency would be a serious bottleneck. A second option is to write a compiler for the language to compile source code down to the machine code, but this would involve redoing work that has already been done for more widely used programming languages. The option we have taken has been to write code generators from PVS to other high-level languages. The compilers for these language can be used to perform low-level optimizations while high-level transformations can be carried out within the logic. The generated code can also be integrated with other, possibly non-critical, code.

[4] Boyer and Moore [BM81] were the first to emphasize executability in the context of theorem provers.

A surprisingly large fraction of higher-order logic turns out to be executable. The executable PVS expressions are those that do not contain any free variables, uninterpreted constants, or equalities between unbounded higher-order types.[5] Any high-level language can serve as the target for code generation from PVS. We have used Common Lisp as the target language since it is also the implementation language for PVS, but several other languages would be equally suitable. Since PVS functions are total, both lazy and eager evaluation would return the same result. Currently, code generation is with respect to an eager order of evaluation. The result of evaluation following code generation from PVS might contain closures, and hence may not be translatable back into PVS. A PVS expression of *non-functional* type, i.e., one that does not contain type components that are of function type, can be evaluated to return a value that is a PVS expression. Executable equality comparisons are those between terms of non-functional type.

Let $[\![e]\!]_\rho$ represent the Common Lisp code generated corresponding to the PVS expression e where ρ binds PVS variables to the corresponding Common Lisp identifiers. A PVS definition of a constant c of type T has the form $c : T = e$. It is translated to `(DEFUN C () E)`, where `E` is $[\![e]\!]_\emptyset$. The primitive PVS arithmetic and logical operations are mapped to the corresponding Lisp operations. The boolean constants `TRUE` and `FALSE` are mapped to `T` and `NIL`, respectively. The integer numerals are mapped to the corresponding Common Lisp numerals. Tuples and records are mapped Common Lisp vectors so that the corresponding projection operations are mapped to vector accesses. Equality on non-functional types is mapped to the `equalp` operation. A lambda-abstraction $\lambda(x : T) : e$ in PVS is mapped to `(FUNCTION (LAMBDA (X) E))`, where `E` is $[\![e]\!]_{\rho\{x \leftarrow X\}}$, and $[\![\mathtt{e}]\!]_{\rho\{\mathtt{x} \leftarrow \mathtt{X}\}}$ is the result of translating the PVS expression e in the context of the free variable x being associated with the Common Lisp variable `X`. PVS conditional expressions are directly mapped to Common Lisp conditional forms.

$$
\begin{aligned}
[\![n]\!]_\rho &= n && \text{for numeral } n\\
[\![\mathtt{TRUE}]\!]_\rho &= \mathtt{T}\\
[\![\mathtt{FALSE}]\!]_\rho &= \mathtt{NIL}\\
[\![(a_1, \ldots, a_n)]\!]_\rho &= (\mathtt{vector}\ [\![a_1]\!]_\rho\ \ldots\ [\![a_n]\!]_\rho)\\
[\![\mathtt{p}_i\ a]\!]_\rho &= (\mathtt{svref}\ [\![a]\!]_\rho\ j), && \text{where } j = i - 1\\
[\![f\ a]\!]_\rho &= (\mathtt{FUNCALL}\ [\![f]\!]_\rho\ [\![A]\!]_\rho)\\
[\![(\mathtt{IF}(a, b, c))]\!]_\rho &= (\mathtt{IF}\ [\![a]\!]_\rho\ [\![b]\!]_\rho\ [\![c]\!]_\rho)\\
[\![(\lambda(x : T) : e)]\!]_\rho &= (\mathtt{FUNCTION}\ (\mathtt{LAMBDA}\ (\mathtt{X})\ \mathtt{E})), && \text{where } \mathtt{E} = [\![e]\!]_{\rho\{x \leftarrow X\}}
\end{aligned}
$$

PVS update expressions are first eliminated prior to translation through the transformations so that the above translation to Common Lisp can be applied to the expression without updates.

$f[(a) := e] = (\lambda j : \mathtt{IF}(j = a, e, f(j))), f$ has function type

[5] PVS 2.3 does not evaluate the full executable subset. It ignores all equalities between higher-order types, and evaluates quantification only over a limited class of types of finite size.

$$r[(a) := e] = (\#l_1 := l_1(r), \ldots, l_i := e, \ldots \#), r \text{ has record type}, a \equiv l_i$$
$$t[(a) := e] = (\mathtt{p}_1\ t, \ldots, e, \ldots), t \text{ has tuple type}, a \equiv i$$

The naïve translation above generates code that is inefficient in a number of ways. Most significantly, the translation of updates through copying causes computations to be grossly inefficient in time and space. Static analysis can be used to identify occurrences of update expressions that can be safely executed in a destructive manner. For simplicity, we restrict attention to one-dimensional array updates of the form $a[i := v]$ occurring in the context of an expression $e[]$. Some examples illustrate the subtleties of destructive update optimizations. We write $f(x_1, \ldots, x_n) = e$ as an alternative form for the definition $f = \lambda(x_1, \ldots, x_n) :\ e$. Let `Arr` be an array type from the subrange `[0..9]` to the integers, i.e., the function type $[[0..9] \rightarrow \mathtt{int}]$. Let A and B be variables of type `Arr`. An array lookup is then written as $A(i)$ and a (nondestructive) array update has the form $A[(c) := d]$. Pointwise addition on arrays $A + B$ is defined to return (a reference to) an array C such that $C(i) = A(i) + B(i)$ for $0 \leq i < 10$. Consider the definitions

$$\begin{aligned} f_1(A) &= A + A[(3) := 4] \\ f_2(A, i) &= A(i) + A[(3) := 4](i) \\ f_3(A) &= A[(3) := f_2(A, 3)] \\ f_4(A, B) &= A + B[(3) := 4] \\ f_5(C) &= f_4(C, C) \end{aligned}$$

When executing $f_1(A)$, the update to A cannot be carried out destructively since the original array is an argument to the $+$ operation. The evaluation of $A[(3) := 4]$ must return a reference to a new array that is a suitably modified copy of the array A. However, in the evaluation $f_2(A)$, given an eager, left-to-right evaluation order, the expression $A(i)$ will be evaluated prior to the update $A[(3) := 4]$. Since the original value of A is no longer used in the computation, the array can be updated destructively.[6] This optimization assumes that array A is not referenced in the context where $f_2(A, i)$ is evaluated. For example, in evaluating $f_3(A)$, it would be incorrect to execute f_2 so that A is updated destructively since there is a reference to the original A in the context when $f_2(A, 3)$ is evaluated. The update to array B in $f_4(A, B)$ can be executed destructively provided A and B are not aliased to the same array reference. Such aliasing happens, for instance, in the evaluation of $f_5(C)$.

The task is that of statically analyzing the definitions of programs involving function definitions such as those of f_1, f_2, f_3, f_4, and f_5, in order to identify those updates that can be executed destructively. Our analysis processes the definition $f(x_1, \ldots, x_n) = e$ of a function f and produces a definition

[6] If a lazy order of evaluation was being employed, this optimization would depend on the order in which the arguments of $+$ were evaluated.

of the form $f^D(x_1, \ldots, x_n) = e^D$, where some occurrences of updates of the form $e_1[(e_2) := e_3]$ in e have been replaced by destructive updates of the form $e_1[(e_2) \leftarrow e_3]$. It also returns a *safety condition* $Sc(f)$ under which it is safe to use f^D instead of f. The safety condition is a partial map from the set of variables $\{x_1, \ldots, x_n\}$ to its power set. If x_i is in the domain of $Sc(f)$, then x_i is possibly destructively updated in f^D, and the variables $Sc(f)(x_i)$ are live in the context of the destructive update. For the examples above, the analysis yields

$$\begin{array}{rll}
f_1^D(A) = & A + A[(3) := 4] & Sc(f_1) = \emptyset \\
f_2^D(A, i) = & A(i) + A[(3) \leftarrow 4](i) & Sc(f_2) = \{A \mapsto \emptyset\} \\
f_3^D(A) = & A[(3) \leftarrow f_2(A, 3)] & Sc(f_3) = \{A \mapsto \emptyset\} \\
f_4^D(A, B) = & A + B[(3) \leftarrow 4] & Sc(f_4) = \{B \mapsto \{A\}\} \\
f_5^D(C) = & f_4(C, C) & Sc(f_5) = \emptyset
\end{array}$$

The update analysis is described in greater detail in unpublished manuscripts [Sha99].[7] We informally describe the analysis for checking when destructive updates are safe in an definition, but do not give the full details on how such updates are identified. The analysis uses three sets of variables that are subsets of a free variable in an expression. The set of *output variables* $O(a)$ consists of the mutable free variables in a that can potentially share structure with the value of a. The set of *active variables* $Ac(a)$ contains $O(a)$ but also includes those mutable variables that may occur in a closure within the value of a. The set of *live variables* $L(e[])$ contain the variables that are potentially live at the point where the hole $[]$ is evaluated. We skip the definitions of these operations and instead illustrate them on the above examples.

1. $f_1(A) = A + A[(3) := 4]$: $Ac(A) = \{A\}$, $O(A) = \{A\}$, and $L(A + []) = \{A\}$. Since the variable A is live in the context $A + []$ and possibly updated in the update expression $A[(3) := 4]$ that fills the hole, it is not safe to translate this update into a destructive one.
2. $f_2(A, i) = A(i) + A[(3) := 4](i)$: $Ac(A(i))$ is $\emptyset$, and hence, $L(A(i) + [](i)$ is also $\emptyset$. Since A, the only variable that is possibly updated in the update expression $A[(3) := 4](i)$, is not live in the context $A(i) + [](i)$, it is safe to transform this update into a destructive one. Since $L(A(i) + [])$ is $\emptyset$, $S(f_2)$ maps A to $\emptyset$.
3. $f_3(A) = A[(3) := f_2(A, 3)]$: $L(A[(3) := []]$ is $\{A\}$, and since f_2^D possibly updates A, the nondestructive version of f_2 must be used. The outer update can be executed destructively since $L([])$ is $\emptyset$, and $Sc(f_3)$ maps A to $\emptyset$.
4. $f_4(A, B) = A + B[(3) := 4]$: $L(A + [])$ is $\{A\}$, and $O(B)$ is $\{B\}$. Since the intersection of $L(A + [])$ and $O(B)$ is empty, the update can be carried out destructively while noting that $Sc(f_4)$ maps B to $\{A\}$.
5. $f_5(C) = f_4(C, C)$: we observe that $Sc(f_4)(B)$ is $\{A\}$, and since C is the actual expression for the formal parameter B, we have that $O(C)$ is $\{C\}$, and

[7] The analysis given here is for the first-order setting with flat arrays. The actual implementation is different in that it deals with a richer language and employs aggressive approximations.

the actual expression for A is C, and $Ac(C)$ is $\{C\}$. Since the intersection is nonempty, it is not safe to invoke the destructive version of f^4 in this definition.

Code generation with the destructive update optimization yields dramatic time and space efficiencies provided the specifier ensures that definitions are written with singly threaded updates. With destructive updates and other high-level optimizations, a functional language can be competitive in performance with the corresponding C program. Currently, the Common Lisp code generated from PVS specifications are around four times slower than the best possible implementations we could carry out of the corresponding C programs, but this gap can easily be narrowed further. If the PVS specifications have been type-checked and all the proof obligations have been discharged, then the only way the generated code can crash during execution is by exceeding a resource bound such as stack space. The PVS type system thus ensures the maximal level of safe execution that can be extracted from any type system for a language that admits unbounded computations.

Draghicescu and Purushothaman [DP93] were the first to use an abstract interpretation in terms of sets of mutable variables but their analysis algorithm is very complex. The destructive update optimization above is similar to one given independently and earlier by Wand and Clinger [WC98].

4 Ground Decision Procedures

We have already seen that higher-order logic with predicate subtypes yields a refined system for typing well-formed expressions including those that generate well-formed computations. Typechecking with predicate subtypes requires the discharging of type-correctness proof obligations. Many typical proof obligations can be discharged automatically through the use of ground decision procedures for equality and arithmetic inequality. Ground decision procedures decide quantifer-free statements, namely those that contain free variables that are implicitly universally quantified at the outermost level. Congruence closure [Koz77,NO80,Sho78,DST80] can be used as a decision procedure for equality for terms built from variables and uninterpreted functions. Most proof obligations contain a combination of uninterpreted function symbols and function symbols that are interpreted within a theory such as linear arithmetic. Nelson and Oppen [NO79] gave a method for combining decision procedures for disjoint theories. Shostak [Sho84] gave a procedure for constructing decision procedures for the combination of solvable and canonizable theories. The PVS decision procedures are based on those of Shostak. In a recently analysis [RS01], we found these procedures to be nonterminating and incomplete, and constructed a sound, complete, and terminating combination algorithm. The corrected decision procedure is described below. It forms the basis of the ICS decision procedures [FORS01] implemented at SRI.

Ground equality decision procedures must verify sequents of the form $T \vdash a = b$ where T is a list of equalities and the free variables in $a = b$ are a subset of those in T. If the function symbols in T, a, and b are uninterpreted, then congruence closure yields a decision procedure. Let $f^3(x)$ abbreviate $f(f(f(x)))$ and $f^5(x)$ abbreviate $f(f(f(f(f(x)))))$. If f is uninterpreted in $f^3(x) = f(x) \vdash f^5(x) = f(x)$, then the equivalence classes over the term universe of sequent generated by the antecedent equality are $\{x\}, \{f(x), f^3(x), f^5(x)\}$, and $\{f^2(x), f^4(x)\}$. This partition clearly validates the conclusion $f^5(x) = f(x)$.

Congruence closure alone will not work with a mix of interpreted and uninterpreted functions, since it will be unable to verify $x + y = z \vdash x = z - y$, for example. Let `FALSE` be an abbreviation for $0 = 1$, then the example

$$f(x-1) - 1 = x + 1,\ f(y) + 1 = y - 1,\ y + 1 = x \vdash \mathtt{FALSE},$$

contains the uninterpreted function f and interpreted operations $+$, $-$, and 1. Shostak's procedure can be used to decide the combination of a canonizable and solvable theory τ with that of equality over uninterpreted function symbols. The conditions of solvability and canonizability are quite technical and a detailed exposition appears elsewhere [RS01]. Roughly, a *canonizable* theory τ is one where there is a canonizer σ so that $\vdash_\tau a = b$ iff $\sigma(a) \equiv \sigma(b)$ for τ-terms a and b. A term a is in canonical form if $\sigma(a) = a$. The term $\sigma(a)$ must always be canonical as must all its subterms, and it must contain no variables that are not already in a. A σ-model is a model M where $M \models a = \sigma(a)$ for any τ-term a, and $M \not\models a = b$ for distinct, canonical, variable-free τ-terms a and b.

The theory τ is *solvable* if there is an operation *solve* such that for any equality $a = b$ between τ-terms, $solve(a = b)$ returns $\bot$ if $a = b$ is unsatisfiable in any σ-model, and otherwise, it returns a *solution* set of equalities S of the form $\{x_1 = t_1, \ldots, x_n = t_n\}$ where each x_i must appear in $a = b$ and not in t_i nor in any other equality $x_j = t_j$ for $i \neq j$. The terms t_j must be canonical. The equality $a = b$ and the set S must be σ-equivalent in the sense that for any assignment ρ, ρ σ-satisfies $a = b$ iff it has some extension ρ' that σ-satisfies S. Note that S might contain variables that are not in $a = b$. Solution sets are essentially idempotent substitutions and can be applied to a term a as $S[a]$ and the composition of substitutions is represented as $S \circ S'$ while noting that the right-hand sides must be kept in σ-normal form.

For a sequent $T \vdash a = b$ containing only τ-terms from a canonizable and solvable theory τ, a decision procedure can be given as follows. Let T be of the form $c_1 = d_1, \ldots, c_n = d_n$. A solution set is computed by the iteration given by

$$\begin{aligned} S_0 &= \emptyset \\ S_{i+1} &= S_i \circ solve(S_i[c_{i+1}] = S_i[d_{i+1}]), \end{aligned}$$

where $S \circ S' = \bot$ if either $S = \bot$ or $S' = \bot$. The sequent is valid if either $S_n = \bot$ or $\sigma(S_n[a]) \equiv \sigma(S_n[b])$. Thus, an assignment ρ σ-satisfies $\bigwedge_{i=1}^n c_i = d_i$ iff it can be extended to a ρ' that σ-satisfies S_n. If $\sigma(S_n[a]) \equiv \sigma(S_n[b])$, then ρ' σ-satisfies

$a = S_n[a] = \sigma(S_n[a]) = \sigma(S_n[b]) = S_n[b] = b$. Otherwise, ρ σ-satisfies T but not $a = b$ and hence $T \vdash a = b$ is not valid.

The basic algorithm gets quite complicated when uninterpreted functions are introduced. A term $f(a_1, \ldots, a_n)$ is said to be uninterpreted if f is uninterpreted. The combined decision procedure must regard uninterpreted terms as variables when solving and canonizing, while identifying congruent uninterpreted terms using congruence closure. Given a solution set S, we define some basic operations. The operation $S\{a\} = \sigma(S[a])$, and $S\langle a\rangle$ returns $S(a)$ if a is a variable, $\sigma(f(S\langle a_1\rangle, \ldots S\langle a_n\rangle))$ if $a \equiv f(a_1, \ldots a_n)$ and f is interpreted, and $S(f(b_1, \ldots, b_n))$ if $a \equiv f(a_1, \ldots, a_n)$, $S(b_i) \equiv S\langle a_i\rangle$ for $0 < i \leq n$, and $f(b_1, \ldots, b_n) \in dom(S)$. The set of subterms of an equality $a = b$ is written as $\llbracket a = b \rrbracket$.

$$\begin{aligned} S_0 &= \emptyset \\ S_{i+1} &= cc(S'_i \circ R), \text{ where} \\ &\qquad \begin{aligned} S'_i &= S \cup \{e = e | e \in \llbracket \hat{a}_{i+1} = \hat{b}_{i+1} \rrbracket - dom(S)\} \\ \hat{a}_{i+1} &= S\langle a_{i+1}\rangle \\ \hat{b}_{i+1} &= S\langle b_{i+1}\rangle \\ R &= solve(\hat{a}_{i+1} = \hat{b}_{i+1}) \end{aligned} \end{aligned}$$

The operation $cc(S)$ performs the congruence closure of S. In each iteration, it picks out a pair of left-hand sides $f(a_1, \ldots, a_n)$ and $f(b_1, \ldots, b_n)$ in S that are congruent with respect to S (f is uninterpreted and $S(a_i) \equiv S(b_i)$ for $0 < i \leq n$), and merges the right-hand sides (when these are distinct) by solving and composing the solution. The *solve* and composition operations treat uninterpreted terms as variables. The soundness, completeness, termination, and complexity of the above procedure are discussed elsewhere [RS01].

Example 1. Consider $f(x-1) - 1 = x + 1,\ f(y) + 1 = y - 1,\ y + 1 = x \vdash$ `FALSE` *from above. Starting with* $S \equiv \emptyset$ *in the base case, the preprocessing of* $f(x - 1) - 1 = x + 1$ *causes the equation to be placed into canonical form as* $-1 + f(-1 + x) = 1 + x$ *and* S *is set to*

$$\{1 = 1, -1 = -1, x = x, -1 + x = -1 + x,\\ f(-1 + x) = f(-1 + x), 1 + x = 1 + x\}.$$

Solving $-1 + f(-1 + x) = 1 + x$ *yields* $f(-1 + x) = 2 + x$, *and* S *is set to*

$$\{1 = 1, -1 = -1, x = x, -1 + x = -1 + x,\\ f(-1 + x) = 2 + x, 1 + x = 1 + x\}.$$

No unmerged congruences are detected. Next, $f(y) + 1 = y - 1$ *is asserted. Its canonical form is* $1 + f(y) = -1 + y$, *and once this equality is asserted, the value of* S *is*

$$\{1 = 1, -1 = -1, x = x, -1 + x = -1 + x,\\ f(-1 + x) = 2 + x, 1 + x = 1 + x, y = y,\\ f(y) = -2 + y, -1 + y = -1 + y, 1 + f(y) = -1 + y\}.$$

Next $y + 1 = x$ is processed. Its canonical form is $1 + y = x$ and the equality $1+y = 1+y$ is added to S. Solving $y+1 = x$ yields $x = 1+y$, and S is reset to

$$\begin{aligned}\{&1 = 1, -1 = -1, x = 1 + y, -1 + x = y,\\ &f(-1 + x) = 3 + y, 1 + x = 2 + y, y = y,\\ &f(y) = -2 + y, -1 + y = -1 + y,\\ &1 + f(y) = -1 + y, 1 + y = 1 + y\}.\end{aligned}$$

The congruence close operation cc detects the congruence $f(1 - y) \overset{S}{\sim} f(x)$ and invokes merge on $3 + y$ and $-2 + y$. Solving this equality $3 + y = -2 + y$ yields $\perp$ returning the desired contradiction.

The ground decision procedures can be used in an incremental manner. Assertions can be added to update the context given by the state of the data structures used by decision procedures. Assertions can be tested for truth or falsity with respect to a given context. In PVS, ground decision procedures are used to build up a context associated with each sequent in a proof consisting of type information and the atomic formulas in the sequent. Many of the leaf nodes of a proof consist of sequents that have been verified by the ground decision procedures. These procedures are also used to simplify expressions using the contextual information given by the governing conditionals and type information from bound variables. The simplifier based on the decision procedures is used to automatically discharge proof obligations that arise in rewriting either through conditions of conditional rewrite rules or through subtype conditions on a matching substitution.

5 Finite-State Methods

Temporal properties of finite-state transition systems form an important decidable class that can be captured elegantly within higher-order logic. A transition system consists of a state type Σ, an initialization predicate I over Σ, and a transition relation N over $[\Sigma, \Sigma]$. Various predicate transformers can be defined as below.

- Pre-image: $pre(N)(p)(s_1) \equiv (\exists s_2 : N(s_1, s_2) \wedge p(s_2))$.
- Pre-condition: $\widetilde{pre}(N)(p)(s_1) \equiv (\forall s_2 : N(s_1, s_2) \supset p(s_2))$.
- Post-condition: $post(N)(p)(s_2) \equiv (\exists s_1 : N(s_1, s_2) \wedge p(s_1))$.

Operators like $pre(N)$, $\widetilde{pre}(N)$, and $post(N)$ are examples of *monotone* predicate transformers, i.e., maps T on $[\Sigma\rightarrow\mathbf{bool}]$ such that

$$p_1 \sqsubseteq p_2 \supset T[p_1] \sqsubseteq T[p_2],$$

where $p_1 \sqsubseteq p_2 \equiv (\forall s : p_1(s) \supset p_2(s))$.

A fixed point of a predicate transformer T is a predicate p such that $p = T[p]$. When T is a monotone predicate transformer, the Tarski-Knaster theorem [Tar55] guarantees the existence of least and greatest fixed points. The least fixed point of a monotone predicate T is written as $\mu X : T[X]$ and can be defined as the greatest lower bound of the predicates X such that $T[X] \sqsubseteq X$, i.e., $\bigcap\{X | T[X] \sqsubseteq X\}$. The greatest fixed point of T is written as $\nu X : T[X]$ and can be defined as the least upper bound of the predicates X such that $X \sqsubseteq T[X]$, i.e., $\bigcup\{X | X \sqsubseteq T[X]\}$.

The temporal connectives of CTL can be defined using the fixed point operators over suitable predicate transformers. For example, the equality

$$[\![\mathbf{A}(B\ \mathbf{U}\ C)]\!](s) = (\mu X : C \vee (B \wedge \widetilde{pre}(N)(X)))(s)$$

holds because the right-hand side is satisfied by exactly those states s that can always reach a state satisfying C along a finite computation path of states satisfying B. The predicate characterizing the reachable states of a transition system K of the form $\langle \Sigma, I, N \rangle$ can be defined as

$$\mu X : I \vee post(N)(X).$$

It is possible to develop the fixed point characterization of the temporal operators in two directions. In one direction, the proof theory of these operators can be derived from the fixed point definitions. In the second direction, automated procedures called model checkers can be developed for verifying temporal properties of transition systems over finite types Σ. The higher-order logic characterization above can be used for both purposes without any change. Thus, model checking and deductive methods are merely different but compatible verification techniques.

The higher-order logic formalization of transition systems and their properties can be used in two ways in model checking. In the symbolic model checking approach, the fixed point iteration is carried out by binary-coding the state so that a truth-value assignment represents a state. The initialization predicate, the transition relation, and temporal properties then become boolean functions. The fixed point iterations can be carried out over some boolean function representation such as reduced, ordered, binary decision diagrams (ROBDD). Another approach, that of bounded model checking, is carried out by encoding the search for a bounded length counterexample to a property as a boolean satisfiability problem. PVS employs an ROBDD-based symbolic model checker [BRB90,BCM+92,Jan93] as a decision procedure for the finite mu-calculus.

The integration of model checking and deductive methods is not all that straightforward. The decision procedures from Section 4 can often be used to automatically discharge the subgoals that arise in an interactive proof, but few of the subgoals in a deductive verification would directly succumb to model checking. Transition system properties can sometimes be verified by building finite-state abstractions that are amenable to model checking. Even when such a finite-state abstraction is not immediately verified by model checking, it is possible to extract useful intermediate properties.

Deductive methods have difficulties with control-intensive systems where abstraction and model checking are quite successful. Invariants are properties of transition systems that are satisfied in all reachable states. Deduction requires an inductive invariant that is preserved by each transition. Finding inductive invariants is not easy. Model checking can restrict its exploration to the reachable states and thus avoids constructing inductive invariants. Abstraction is a useful way of decomposing the verification task between deductive and explorative methods. An abstraction mapping indicates how the concrete state is mapped to a finite state. In the case of predicate abstraction, the abstract state contains booleans that capture predicates on the concrete state. An abstract model approximating the behavior of the concrete transition system can be constructed by using a theorem prover to eliminate abstract transitions that have no concrete counterpart. Then the reachable states of the resulting abstract transition system can be used to construct a valid invariant for the concrete transition system.

Ground decision procedures play a crucial role in the construction of an abstract transition system from a concrete one based on a given abstraction map. This construction involves a large number of small-sized proof obligations that are mostly dischargeable using decision procedures.

Predicate abstraction was first used by Graf and Saïdi [SG97] to construct the abstract reachability graph of a transition system with respect to an abstraction map. The InVest system [BLO98] uses the elimination method mentioned above to construct abstractions of transition systems described in a fragment of PVS. An abstraction capability for the mu-calculus has been added to PVS in order to abstract transition systems along with their temporal properties [SS99,Sha01].

6 Discussion

We have barely scratched the surface of the kinds of problems that can be naturally expressed and effectively automated using higher-order logics. As a pragmatic point, we have argued that higher-order logic provides a good balance between expressiveness and automation by embedding useful formalisms for which there is good automation support. Most of these problems can already be expressed in second-order or third-order logic, but the full range of finite-order types adds very little complexity. The directness with which these logics can be embedded in higher-order logic allows the decision procedures to be used without having to span the semantic gap between diverse logics.

We have shown a few ways in which decision procedures interact synergistically with higher-order logic. They can be used to add expressivity to the type system through predicate subtypes and dependent types. Static analysis can be applied to higher-order programs to infer safe destructive array updates. The resulting optimized programs can be executed with efficiencies that are comparable with implementations in low-level imperative programming languages. Fixed point calculi can be embedded in higher-order logic. A number of temporal logics can be defined using the fixed point operators. The finite-state version of the fixed point calculus can be decided using a BDD-based symbolic model checker. Ground decision procedures can be used to construct finite-state abstractions from large or infinite-state transition systems. Decidable fragments of higher-order logic can be used to verify properties of parametric systems.

The synergy between expressiveness and automation can be further explored along a number of lines. We have examined evaluation in a functional sublanguage of higher-order logic. A number of high-level optimizations can be explored in the context of a theorem proving environment. It is also possible to embed programming vernaculars such as logic programming [Sym98] and imperative programming. Higher-order logic programming with the use of higher-order syntactic encoding has emerged as an important medium for metaprogramming [NM90]. A number of useful decision procedures from computer algebra, operations research, and engineering, can also be usefully incorporated into a theorem proving environment. Drawing inspiration from the work of Boyer and Moore [BM81], a metaprogramming capability can be used to reflectively develop a theorem proving system within itself in a bootstrapped manner.

Proof search is an area where PVS is currently inadequate. First-order proof search methods have been successfully used in Isabelle [Pau98]. Proof search methods within the full higher-order logic have had some selective successes [AINP88,BK98], but quantifier-elimination methods and proof search methods for limited fragments are more likely to yield profitable results in the short run.

The entire enterprise of theorem proving in higher-order logics is poised at an interesting confluence between expressiveness and automation, principle and pragmatism, and theory and practice. Significant breakthroughs can be anticipated if we see synergies rather than conflict in this confluence.

Acknowledgments. I thank the TPHOLS'01 conference chair Richard Boulton, the programme chair Paul Jackson, and the programme committee, for their kind invitation. I am also grateful to the TPHOLS community for stimulus and inspiration. Michael Kohlhase was generous with his insights on higher-order proof search. Paul Jackson gave me detailed and helpful comments on an earlier draft, as did Sam Owre and John Rushby. The ideas and results presented here are the fruits of collaborations with several people including Sam Owre, John Rushby, Sree Rajan, Harald Rueß, Hassen Saïdi, and Mandayam Srivas.

References

[AINP88] Peter B. Andrews, Sunil Issar, Daniel Nesmith, and Frank Pfenning. The TPS theorem proving system. In E. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction (CADE)*, volume 310 of *Lecture Notes in Computer Science*, pages 760–761, Argonne, IL, May 1988. Springer-Verlag.

[And86] Peter B. Andrews. *An Introduction to Logic and Type Theory: To Truth through Proof.* Academic Press, New York, NY, 1986.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[BK98] Christoph Benzmüller and Michael Kohlhase. Extensional higher-order resolution. In H. Kirchner and C. Kirchner, editors, *Proceedings of CADE-15*, number 1421 in Lecture Notes in Artificial Intelligence, pages 56–71, Berlin, Germany, July 1998. Springer-Verlag.

[BLO98] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In Hu and Vardi [HV98], pages 319–331.

[BM81] R. S. Boyer and J S. Moore. Metafunctions: Proving them correct and using them efficiently as new proof procedures. In R. S. Boyer and J S. Moore, editors, *The Correctness Problem in Computer Science*. Academic Press, London, 1981.

[BM86] R. S. Boyer and J S. Moore. Integrating decision procedures into heuristic theorem provers: A case study with linear arithmetic. In *Machine Intelligence*, volume 11. Oxford University Press, 1986.

[Bou93] R. J. Boulton. Lazy techniques for fully expansive theorem proving. *Formal Methods in System Design*, 3(1/2):25–47, August 1993.

[BRB90] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proc. of the 27th ACM/IEEE Design Automation Conference*, pages 40–45, 1990.

[Bun83] Alan Bundy. *The Computer Modelling of Mathematical Reasoning.* Academic Press, London, UK, 1983.

[But97] Bettina Buth. PAMELA + PVS. In Michael Johnson, editor, *Algebraic Methodology and Software Technology, AMAST'97*, volume 1349 of *Lecture Notes in Computer Science*, pages 560–562, Sydney, Australia, December 1997. Springer-Verlag.

[CAB+86] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, Englewood Cliffs, NJ, 1986.

[CDC80] Mario Coppo and Mariangiola Dezani-Ciancaglini. An extension of the basic functionality theory for the lambda-calculus. *Notre Dame J. Formal Logic*, 21(4):685–693, 1980.

[CH85] T. Coquand and G. P. Huet. Constructions: A higher order proof system for mechanizing mathematics. In *Proceedings of EUROCAL 85, Linz (Austria)*, Berlin, 1985. Springer-Verlag.

[Chu40] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[dB80] N. G. de Bruijn. A survey of the project Automath. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 589–606. Academic Press, 1980.

[DCN+00] Louise A. Dennis, Graham Collins, Michael Norrish, Richard Boulton, Konrad Slind, Graham Robinson, Mike Gordon, and Tom Melham. The PROSPER toolkit. In Susanne Graf and Michael Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, number 1785 in Lecture Notes in Computer Science, pages 78–92, Berlin, Germany, March 2000. Springer-Verlag.

[DFH+91] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Christine Paulin-Mohring, and Benjamin Werner. The COQ proof assistant user's guide: Version 5.6. Rapports Techniques 134, INRIA, Rocquencourt, France, December 1991.

[DKML99] Martin Dunstan, Tom Kelsey, Ursula Martin, and Steve Linton. Formal methods for extensions to CAS. In Jeannette Wing and Jim Woodcock, editors, *FM99: The World Congress in Formal Methods*, volume 1708 and 1709 of *Lecture Notes in Computer Science*, pages 1758–1777, Toulouse, France, September 1999. Springer-Verlag. Pages 1–938 are in the first volume, 939–1872 in the second.

[DP93] M. Draghicescu and S. Purushothaman. A uniform treatment of order of evaluation and aggregate update. *Theoretical Computer Science*, 118(2):231–262, September 1993.

[DST80] P.J. Downey, R. Sethi, and R.E. Tarjan. Variations on the common subexpressions problem. *Journal of the ACM*, 27(4):758–771, 1980.

[Fef78] Solomon Feferman. Theories of finite type related to mathematical practice. In Jon Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, chapter D4, pages 913–972. North-Holland, Amsterdam, Holland, 1978.

[FORS01] J-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated canonizer and solver. In *CAV 01: Computer-Aided Verification*. Springer-Verlag, 2001. To appear.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, UK, 1993.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[Gor88] M. J. C. Gordon. Mechanizing programming logics in higher order logic. Technical Report CCSRC-006, Cambridge Computer Science Research Center, SRI International, Cambridge, England, September 1988.

[Har96] John Harrison. Stålmarck's algorithm as a HOL derived rule. In J. von Wright, J. Grundy, and J. Harrison, editors, *Proceedings of the 9th TPHOLS*, number 1125 in Lecture Notes in Computer Science, pages 251–266, Berlin, Germany, 1996. Springer-Verlag.

[Hat68] William S. Hatcher. *Foundations of Mathematics*. W. B. Saunders Company, Philadelphia, PA, 1968.

[HD92] F. K. Hanna and N. Daeche. Dependent types and formal synthesis. In C. A. R. Hoare and M. J. C. Gordon, editors, *Mechanized Reasoning and Hardware Design*, pages 121–135, Hemel Hempstead, UK, 1992. Prentice Hall International Series in Computer Science.

[HV98] Alan J. Hu and Moshe Y. Vardi, editors. *Computer-Aided Verification, CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, Vancouver, Canada, June 1998. Springer-Verlag.

[HWG98] David Hardin, Matthew Wilding, and David Greve. Transforming the theorem prover into a digital design tool: From concept car to off-road vehicle. In Hu and Vardi [HV98], pages 39–44.

[Jan93] G. L. J. M. Janssen. *ROBDD Software*. Department of Electrical Engineering, Eindhoven University of Technology, October 1993.

[JvdBH+98] Bart Jacobs, Joachim van den Berg, Marieke Huisman, Martijn van Berkum, Ulrich Hensel, and Hendrick Tews. Reasoning about Java classes. In *Proceedings, Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, pages 329–340, Vancouver, Canada, October 1998. Association for Computing Machinery. Proceedings issued as ACM SIGPLAN Notices Vol. 33, No. 10, October 1998.

[Koz77] D. Kozen. Complexity of finitely represented algebras. In *Proc. 9th ACM STOC*, pages 164–177, 1977.

[Lei94] Daniel Leivant. Higher order logic. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 2: Deduction Methodologies*, pages 229–321. Clarendon Press, Oxford, 1994.

[NL00] George C. Necula and Peter Lee. Proof generation in the touchstone theorem prover. In David McAllester, editor, *Automated Deduction—CADE-17*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 25–44, Pittsburgh, PA, June 2000. Springer-Verlag.

[NM90] G. Nadathur and D. Miller. Higher-order Horn clauses. *Journal of the ACM*, 37(4):777–814, 1990.

[NO79] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[NO80] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, 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 (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[ORSvH95] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[OS97] Sam Owre and Natarajan Shankar. The formal semantics of PVS. Technical Report SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1997.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[Pau98] L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1):85–128, 1998.

[Ram90] F. P. Ramsey. The foundations of mathematics. In D. H. Mellor, editor, *Philosophical Papers of F. P. Ramsey*, chapter 8, pages 164–224. Cambridge University Press, Cambridge, UK, 1990. Originally published in *Proceedings of the London Mathematical Society*, 25, pp. 338–384, 1925.

[ROS98] John Rushby, Sam Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, September 1998.

[RS01] Harald Rueß and Natarajan Shankar. Deconstructing Shostak. In *Proceedings 16th Annual IEEE Symp. on Logic in Computer Science*, pages 19–28. IEEE Computer Society Press, 2001.

[RvHO91] John Rushby, Friedrich von Henke, and Sam Owre. An introduction to formal specification and verification using EHDM. Technical Report SRI-CSL-91-2, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1991.

[SG97] Hassen Saïdi and Susanne Graf. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Computer-Aided Verification, CAV '97*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, June 1997. Springer-Verlag.

[Sha99] N. Shankar. Efficiently executing PVS. Project report, Computer Science Laboratory, SRI International, Menlo Park, CA, November 1999. Available at http://www.csl.sri.com/shankar/PVSeval.ps.gz.

[Sha01] Natarajan Shankar. Automated verification using deduction, exploration, and abstraction. In *Essays on Programming Methodology*. Springer-Verlag, 2001. To appear.

[Sho78] R. Shostak. An algorithm for reasoning about equality. *Comm. ACM*, 21:583–585, July 1978.

[Sho84] R. E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, 1984.

[SS99] Hassen Saïdi and N. Shankar. Abstract and model check while you prove. In Nicolas Halbwachs and Doron Peled, editors, *Computer-Aided Verification, CAV '99*, volume 1633 of *Lecture Notes in Computer Science*, pages 443–454, Trento, Italy, July 1999. Springer-Verlag.

[Sym98] D. Syme. *Declarative Theorem Proving for Operational Semantics*. PhD thesis, University of Cambridge, 1998.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. of Math.*, 5:285–309, 1955.

[Thé98] Laurent Théry. A certified version of Buchberger's algorithm. In H. Kirchner and C. Kirchner, editors, *Proceedings of CADE-15*, number 1421 in Lecture Notes in Artificial Intelligence, pages 349–364, Berlin, Germany, July 1998. Springer-Verlag.

[vBD83] Johan van Benthem and Kees Doets. Higher-order logic. In Dov M. Gabbay and Franz Guenthner, editors, *Handbook of Philosophical Logic–Volume I: Elements of Classical Logic*, volume 164 of *Synthese Library*, chapter I.4, pages 275–329. D. Reidel Publishing Company, Dordrecht, Holland, 1983.

[vHCL+88] F. W. von Henke, J. S. Crow, R. Lee, J. M. Rushby, and R. A. Whitehurst. The EHDM verification environment: An overview. In *Proceedings 11th National Computer Security Conference*, pages 147–155, Baltimore, MD, October 1988. NBS/NCSC.

[WC98] Mitchell Wand and William D. Clinger. Set constraints for destructive array update optimization. In *Proc. IEEE Conf. on Computer Languages '98*, pages 184–193. IEEE, April 1998.

[WR27] A. N. Whitehead and B. Russell. *Principia Mathematica*. Cambridge University Press, Cambridge, revised edition, 1925–1927. Three volumes. The first edition was published 1910–1913.