TPHOLS talk

Using Decision Procedures With a Higher-Order Logic

N. Shankar[a]

shankar@csl.sri.com

URL: http://www.csl.sri.com/~shankar/

Computer Science Laboratory

SRI International

Menlo Park, CA

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

CAVman versus CADEman

CAVman: I can't seem to get your theorem prover to verify a *%#$ thing?

CADEman: My theorem prover is complete. It can, in principle, prove arbitrary theorems.

CAVman: But I have a specific problem for which my method works automatically and efficiently. When was the last time you actually proved a theorem?

CADEman: I can verify most of the TPTP library. My program has even settled long-standing open problems.

CAVman: Your open problem is that you don't have a technology.

CADEman: Your problems are intellectually trivial and your methods unscientific.

Main Thesis
Micrologic versus Macrologic

- Micrologic addresses individual problems emphasizing automation. (CAV)
- Macrologic deals with the logic as a whole using problem-independent deduction techniques. (CADE)
- Macrologic is "interesting", but micrologic is what matters.
- Higher order logic has *macrological* disadvantages, but is effective at supporting micrologic through
  - Direct embedding of micrological formalisms, and
  - Smooth integration of special-purpose deductive procedures.

A Computational Sapir–Whorf Hypothesis

Language/formalism influences automated reasoning.

First-order versus Higher-order Logic

The choices for formalization are:

- First-order logic with *ad hoc* non-logical axioms and axiom schemes. Good for proof search in mathematically (finite, first-order axiomatizable) weak theories.

- Higher-order logic, possibly with axioms. Good for expression and manipulation.

Completeness of first-order logic versus incompleteness of higher-order logic is a non-issue.

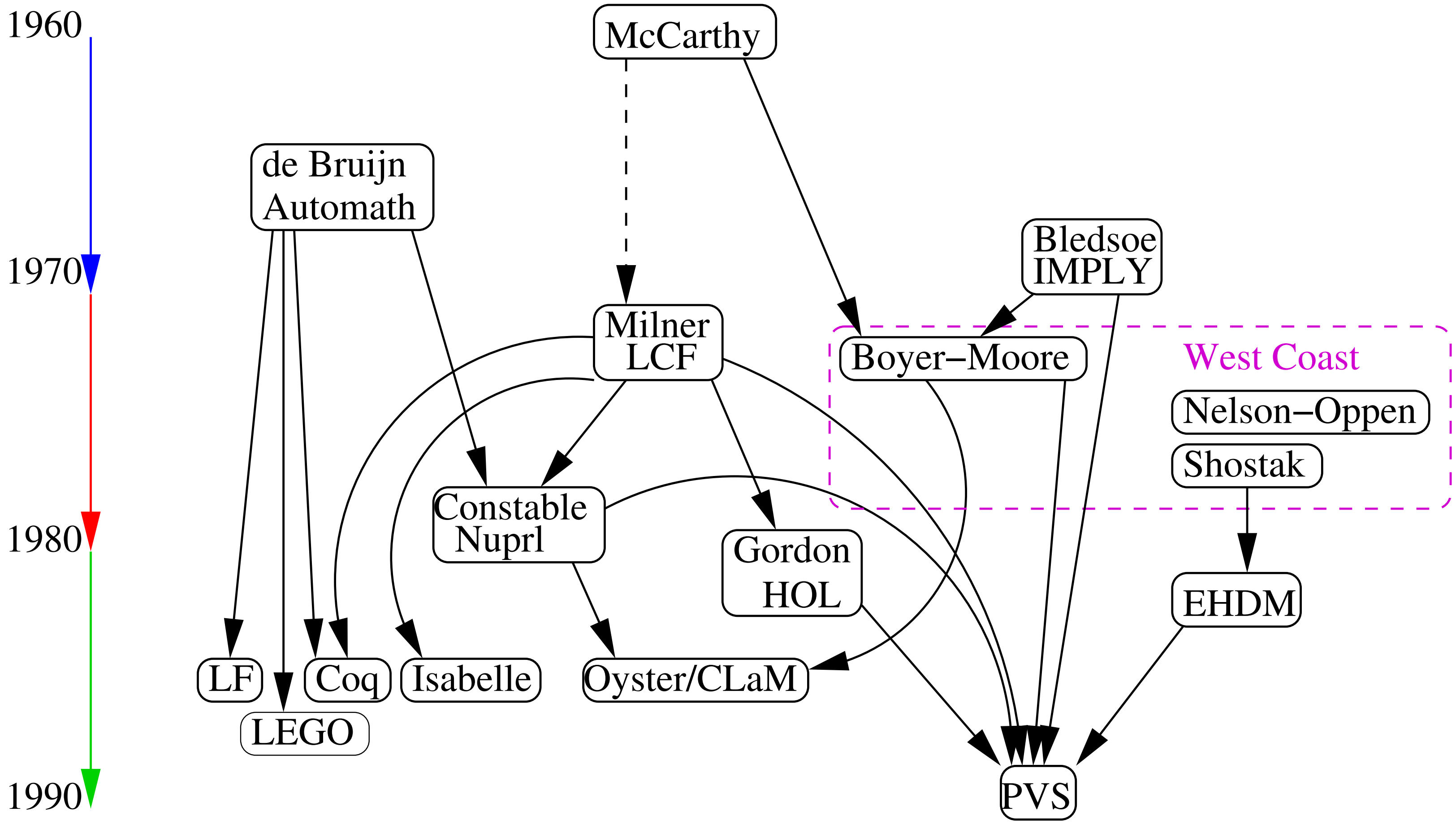
Higher-order logic captures transitive closure, fixed points, induction, finiteness.

First-order automation is compatible with higher-order logic.

Outline

- Historical Background: Where does PVS come from?
- PVS language
- Ground decision procedures
- Evaluation
- Model checking and abstraction
- Conclusions

History


1960
1970
1980
1990
McCarthy
de Bruijn Automath
Bledsoe IMPLY
Milner LCF
Boyer–Moore
West Coast
Nelson–Oppen
Shostak
Constable Nuprl
Gordon HOL
EHDM
LF
Coq
Isabelle
Oyster/CLaM
LEGO
PVS

The Origins of PVS

- Jeff Joyce (then at Cambridge) visited SRI in 1989/90.
- We had a bake-off between HOL, Nqthm, and EHDM.
  Nqthm won!
- Jeff was not convinced.
  His response: HOL users like the "illusion of progress."
- Lightbulbs went off. We realized that we needed to work on wrapping automation within an illusion of progress.
- John Rushby somehow acquired SRI funding, and Sam Owre worked his system-building magic.

PVS Design Desiderata

- A People's Verification System: Take theorem proving out of the priesthood.
- A rich specification language building on EHDM.
- Good balance between interaction and automation
  - Results of automation should be predictable and readable
  - User is just another proof strategy.
- A Prototype Verification System: Shoot first, ask questions later.

PVS Language: Predicate Subtypes

Based on a suggestion of Friedrich von Henke (inspired by ANNA) that was implemented in Ehdm.

```
even?(i): bool = EXISTS j: i = j * 2

even_int: NONEMPTY_TYPE = (even?) CONTAINING 0

half(x : even_int): RECURSIVE {y :int | 2*y = x} =
  (IF x = 0
   THEN 0
   ELSIF x > 0
   THEN half(x - 2) + 1
   ELSE half(x + 2) - 1
  ENDIF)
MEASURE abs(x)
```

Proof Obligations: Subtypes

Typechecking generates subtype TCCs (Type Correctness Conditions).

```
half(x : even_int): RECURSIVE {y :int | 2*y = x} =
  (IF x = 0
   THEN 0
   ELSIF x > 0
   THEN half(x - 2) + 1
   ELSE half(x + 2) - 1
   ...
```

```
half_TCC1: OBLIGATION FORALL (x: even_int): x = 0 IMPLIES 2 * 0 = x;

half_TCC3: OBLIGATION
  FORALL (v: [x: even_int -> {y: int | 2 * y = x}], x: even_int):
    NOT x = 0 AND x > 0 IMPLIES 2 * (v(x - 2) + 1) = x;
  ...
```

Proof Obligations: Termination

```
half(x : even_int): RECURSIVE {y :int | 2*y = x} =
  (IF x = 0
   THEN 0
   ELSIF x > 0
   THEN half( x - 2 ) + 1
   ELSE half( x + 2 ) - 1
  ENDIF)
MEASURE abs(x)
```

```
half_TCC2: OBLIGATION
  FORALL (x: even_int): NOT x = 0 AND x > 0
                 IMPLIES abs(x - 2) < abs(x);

half_TCC4: OBLIGATION
  FORALL (x: even_int): NOT x = 0 AND NOT x > 0
                    IMPLIES abs(x + 2) < abs(x);
```

Missing Proof Obligations

```
half(x : even_int): RECURSIVE {y :int | 2*y = x} =
  (IF x = 0
   THEN 0
   ELSIF x > 0
   THEN half(x - 2) + 1
   ELSE half(x + 2) - 1
  ENDIF)
MEASURE abs(x)
```

Why don't we have subtype proof obligations corresponding to `x - 2 : even_int` and `x + 2 : even_int`?

Typing Judgements

Judgements are used to record refined type information that can be propagated by the typechecker.

Allows TCC proof obligations to be proved once and for all.

```
e1, e2: VAR even_int

even_plus_even_is_even:  JUDGEMENT +(e1,e2) HAS_TYPE even_int

even_minus_even_is_even: JUDGEMENT -(e1,e2) HAS_TYPE even_int
```

Dependent Typing

Inspired by difficulties experienced by Carl Witty when implementing stacks. half is also dependent.

```
astack [T : TYPE] : THEORY
  BEGIN
   astack : TYPE = [# size : nat,
                      elems: [below(size) -> T] #]
   S, S0, S1: VAR astack
   a, b, c: VAR T

   empty?(S): bool = (S`size = 0)
   nonempty?(S): bool = NOT empty?(S)
   nonempty_stack: TYPE = (nonempty?)

   push(a, S): nonempty_stack =
     S WITH [`size |-> S`size + 1,
             `elems(S`size) |-> a]
  END astack
```

Summarizing

- A predicate subtype is a subtype (subset) consisting of elements of a given type that satisfy a given predicate.
- Typechecking generates proof obligations corresponding to predicates, but simple type correctness is simple.
- Typechecking is undecidable, but only modulo the discharging of proof obligations.
- Dependent typing: Predicate in one component of a type depends on the value of another component.
- Typical proof obligations can be discharged automatically using decision procedures.
- Conversely, subtype information can be exploited by the decision procedures.

Ground Decision Procedures

Ground Decision Procedures (GDP)

GDPs are the hallmark of *west coast* theorem proving.

Pioneered by Nelson and Oppen in the Stanford Pascal Verifier and Shostak in STP.

Used by PVS, SVC, STeP.

In PVS, Shostak's decision procedure

1. Builds logical context
2. Discharges proof obligations (from typechecking, rewriting, and abstraction)
3. Completes leaf nodes of proofs

Historical Background on Shostak

Shostak's implementation from early 80's (heavily patched) is still the core of the PVS.

Cyrluk, Lincoln, Shankar (1996) gave an exposition of Shostak's method with some corrections, and sketched soundness and completeness

Shostak's procedure has also been adapted and extended by SVC [Barrett, Dill, Levitt, 1996] and STeP [Bjørner, 1999].

All of these algorithms are flawed as described. They are *incomplete* and potentially *non-terminating*.

GDPs for Equality in Combinations of Theories

Decide sequents of the form $T \vdash A$.

1. $x = y,\ z = y \vdash z = x$, transitivity.
2. $x = f(x) \vdash f(f(x)) = x$, congruence. f is uninterpreted.
3. $f(f(f(x))) = f(x) \vdash f^5(x) = f(x)$.
4. $f(u+1) - 1 = v + 1,\ f(w-1) = u,\ u + 2 = w \vdash w - 1 = v + 2$.
   Linear arithmetic with uninterpreted function symbols.
5. $f(v) = v, f(u) = u - 1, u = v \vdash 0 = 1$.
   We abbreviate $0 = 1$ as false.
6. $f(x-1) - 1 = x + 1,\ f(y) + 1 = y - 1,\ y + 1 = x \vdash \mathit{false}$.

Nelson–Oppen Combination

Consider $f(x-1)-1=x+1,\ f(y)+1=y-1,\ y+1=x \vdash \mathit{false}$.

Variable abstraction: New variables u, w, z, such that $u = x-1, w = f(u), z = f(y)$.

	Q	L
Given	$w = f(u)$ $z = f(y)$	$u = x-1$ $w-1 = x+1$ $z+1 = y-1$ $y+1 = x$
By L		$u = y$
By Q	$w = z$	
By L		$x = y-4 = y+1$
By L		Contradiction

Congruence closure

Decides equality with uninterpreted function symbols.

Sequent $f(f(f(x))) = f(x) \vdash f^5(x) = f(x)$ generates a congruence-closed partition

$$\{x\}, \quad \{f(x), f(f(f(x))), f^5(x)\}, \quad \{f(f(x)), f^4(x)\}$$

of its term universe.

Congruence closure algorithms studied by Kozen (1977), Nelson and Oppen (1977), Shostak (1978), and Downey, Sethi, and Tarjan (1980).

Shostak's Congruence Closure

Online algorithm: Decides $T \vdash A$ by processing T and checking A.

Processing $f(f(f(x))) = f(x)$ yields solution set $S = \{x \mapsto x, f(x) \mapsto f^3(x), f(f(x)) \mapsto f(f(x)), f^3(x) \mapsto f^3(x)\}$.

$can(S)(a)$: the canonical form of a w.r.t. S.

$$
\begin{array}{rcll}
can(S)(x) & = & S(x) & \\
can(S)(f(a)) & = & S(f(b)), \text{ when } b: & f(b) \in dom(S), \\
 & & & S(b) \equiv can(S)(a). \\
can(S)(f(a)) & = & f(can(S)(a)), \text{ otherwise.} &
\end{array}
$$

E.g., $can(S)(f^5(x)) \equiv f(f(f(x))) \equiv can(S)(f(x))$.

Shostak's Combination Procedure

Combines interpreted functions from a *canonizable*, *solvable* theory with equality over uninterpreted function symbols.

A canonizable theory has a canonizer σ such that

$$\vdash a = b \iff \sigma(a) \equiv \sigma(b).$$

A σ-model M is one where $M \models a = \sigma(a)$.

A solvable theory has a solver $solve$ such that

$solve(a = b)$ returns $\bot$, if $a = b$ is σ-unsatisfiable, and

$solve(a = b)$ returns a solved form S such that
$\sigma(S[a]) \equiv \sigma(S[b])$. ($S[a]$ applies S as a substitution)

Define $can(S)(f(a)) = \sigma(f(can(S)(a)))$, f interpreted.

Shostak as Online Gaussian Elimination

- To prove $T \vdash a = b$, where T, a, b, contain pure theory terms from a canonizable and solvable theory:
- Let T be of the form $\{c_1 = d_1, \ldots, c_n = d_n\}$:

$$
\begin{array}{rcl}
S_0 & = & \emptyset \\
S_{i+1} & = & S_i \circ solve(S_i[c_{i+1}] = S_i[d_{i+1}]) \\
\perp \circ S' & = & \perp \\
S \circ \perp & = & \perp \\
S \circ S' & = & \{l \mapsto \sigma(S'[r]) \mid l \mapsto r \in S\} \cup S'
\end{array}
$$

- $T \vdash a = b$ is valid if either $S_n = \perp$ or $\sigma(S_n[a]) \equiv \sigma(S_n[b])$.

Canonizable and Solvable Theories

Lots of theories are canonizable and solvable

1. Uninterpreted equality
2. Linear arithmetic: ordered sum-of-products, Gaussian elimination
3. Recursive datatypes
4. Set algebra
5. Bit-vectors
6. Arrays
7. Propositional Logic

A Combined Decision Procedure

Let σ and *solve* treat uninterpreted subterms as variables.

To prove $T \vdash a = b$ for $T = \{c_1 = d_1, \ldots, c_n = d_n\}$: [a]

$$
\begin{array}{rcl}
S_0 & = & \emptyset \\
S_{i+1} & = & cc(merge(can(S_i)(c), can(S_i)(d), S_i)) \\
merge(c, d, S) & = & S \circ solve(c = d) \\
cc(S) & = & cc(merge(r_1, r_2, S)), \\
 & & \text{when } l_1, l_2 : l_1 \overset{S}{\sim} l_2 \\
 & & S(l_1) \equiv r_1, S(l_2) \equiv r_2, r_1 \not\equiv r_2 \\
cc(S) & = & S, \text{ otherwise.}
\end{array}
$$

Let $S = S_n \neq \bot$, then $can(S)(a) \equiv can(S)(b)$ iff $T \vdash a = b$ is derivable iff $T \models a = b$ in all σ-models.

[a] Algorithm given is simplified and slightly inaccurate.

Example

For $f(x-1)-1 = x+1, \quad f(y)+1 = y-1, \quad y+1 = x \vdash \mathit{false}$.

$S_0 = \emptyset$

$$can(S_0)(f(x-1)-1) \equiv f(x-1)-1; \; can(S_0)(x+1) \equiv x+1$$

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

$S_1 = \{x \mapsto x, \; f(x-1) \mapsto x+2, x-1 \mapsto x-1\}$

$$can(S_1)(f(y)+1) \equiv f(y)+1; \; can(S_1)(y-1) \equiv y-1$$

$$solve(f(y)+1 = y-1) = \{f(y) = y-2\}$$

$S_2 = \{x \mapsto x, \; f(x-1) \mapsto x+2, \; y \mapsto y, f(y) \mapsto y-2, x-1 \mapsto x-1\}$

$$can(S_2)(y+1) \equiv y+1; \quad can(S_2)(x) = x$$

$$solve(y+1 = x) = \{y = x-1\}$$

$$S_3 = \left\{ \begin{array}{l} x \mapsto x, \; f(x-1) \mapsto x+2, \; y \mapsto x-1, \\ f(y) \mapsto x-3, \; x-1 \mapsto x-1 \end{array} \right\}$$

Since $f(x-1) \overset{S_3}{\approx} f(y)$, invoke $solve(x+2 = x-3) = \bot$.

ICS Decision Procedures

The algorithm and proof are described in *Deconstructing Shostak* (LICS 2001, with Rueß).

It forms the basis for ICS (Integrated Canonizer and Solver) implemented with a rich API in oCaml (by Harald Rueß following an initial implementation by Jean-Christophe Filliâtre).

ICS is available currently under a noncommercial license from `www.icansolve.com`.

Will be included with PVS3.0 (late 2001 or early 2002).

Jonathan Ford is attempting to verify the algorithm and proofs from *Deconstructing Shostak* in PVS.

Evaluating PVS

Evaluating PVS

Motivated by demands from Rockwell-Collins engineers for a validation capability (emulating Nqthm/ACL2).

Code generation from PVS for animation and validation.

Current target is Common Lisp, but other targets are also interesting.

Only non-executable form is equality between functions over unbounded domains (e.g., quantification), but PVS executable subset is more limited.

If the code generation is correct and all TCCs have been proved, resulting code can be safely run without runtime checks. Crashes only when out of a resource.

Safe Destructive Optimization

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

Functional languages can be executed with time/space efficiencies competitive with C.

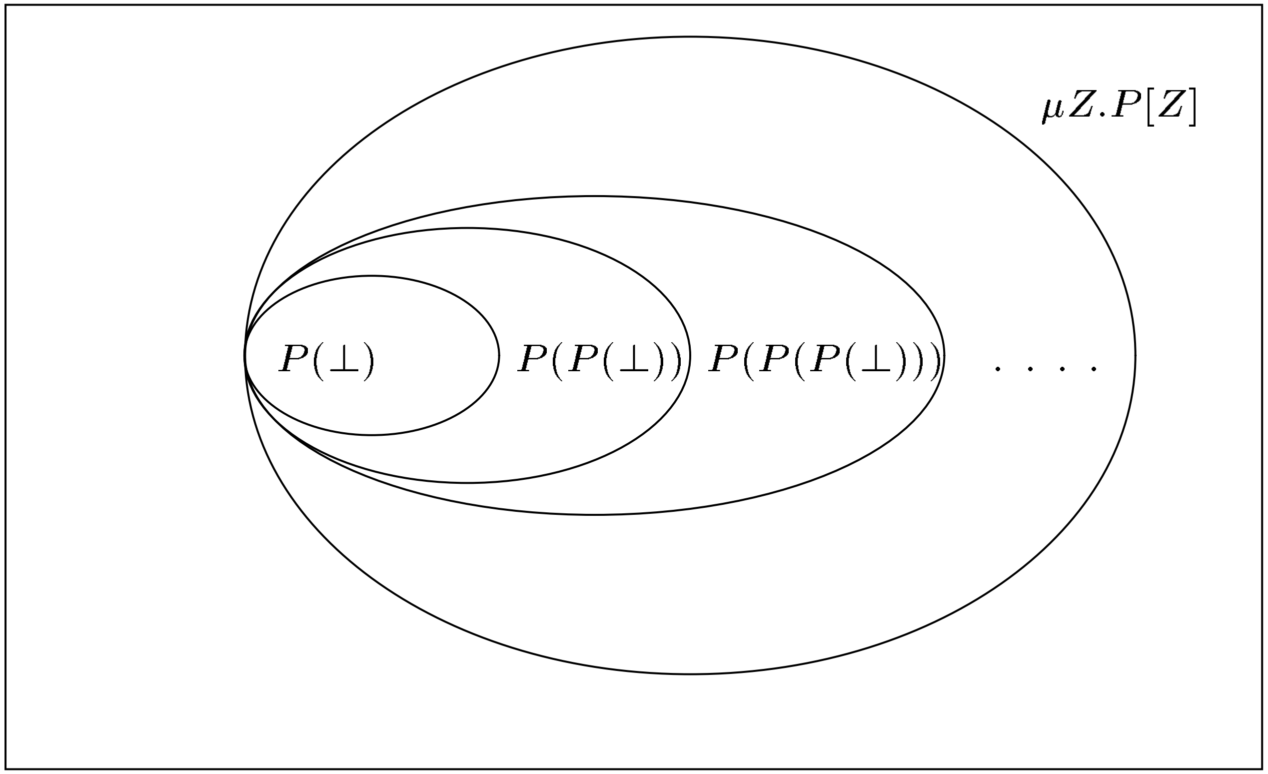
Similar analysis given independently and earlier by Wand and Clinger.

Model Checking and Abstraction

Predicate Transformers

- - 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:
    $post(N)(p)(s_2) \equiv (\exists s_1 : N(s_1, s_2) \wedge p(s_1))$.
- Given a monotone predicate transformer F, define
  - Least fixed point: $\bigcap\{X \mid F(X) \subseteq X\}$.
  - Greatest fixed point: $\bigcup\{X \mid X \subseteq F(X)\}$.
- *Definable* in higher-order logic [Rajan, S, Srivas, CAV'95].

The Least Fixed Point



Fixed Point Computations

The set of reachable states is fundamental to model checking

- Any initial state is reachable.
- Any state that can be reached in a single transition from a reachable state is reachable.
- These are all the reachable states.

This is a least fixed point: $\mu Q : I \vee post(N)(Q)$.

An invariant is an assertion that is true of all reachable states:

$$\mathbf{AG_N} P = \neg \mu Q : \neg P \vee pre(N)(Q)$$

Symbolic Fixed Point Computations

If the computation state is represented as a boolean array $b[1..N]$,

Then a set of states can be represented by a boolean function mapping $\{0, 1\}^N$ to $\{0, 1\}$.

Boolean functions can represent:

- Initial state set
- Transition relation
- Image of transition relation with respect to a state set

Set of reachable states computable as a boolean function.

ROBDD representation of boolean functions empirically efficient.

Deduction Versus Model Checking

- Deduction is general but requires invariant strenthening and manual guidance. Suffers from case explosion.
- Model checking is applicable to small finite-state systems. Suffers from state explosion.
- Model checking does not require invariant strengthening.
- But many systems have large or infinite state spaces.
- Property preserving abstractions can be used to reduce such systems to model checkable form.
- Predicate abstraction [Graf/Saïdi] is a powerful technique for combining deduction and model checking.

A Bakery-like Algorithm for Mutual Exclusion

	P_1	P_2
init	$y_1 = 0$	$y_2 = 0$
loop		
sleep	$y_1 := y_2 + 1;$	$y_2 := y_1 + 1;$
try	**await** $y_2 = 0 \vee y_1 < y_2;$	**await** $y_1 = 0 \vee y_2 \leq y_1;$
critical	$y_1 := 0$	$y_2 := 0$
abstract	$a \equiv y_1 = 0;\ b \equiv y_2 = 0;\ c \equiv y_1 < y_2$	

	Q_1	Q_2
init	$a = \mathtt{T}$	$b, c = \mathtt{T}, \mathtt{F}$
loop		
`sleep`	$a, c := \mathtt{F}, \mathtt{F}$;	$b, c := \mathtt{F}, \mathtt{T};$
`try`	**await** $b \vee c$;	**await** $a \vee \neg c$;
`critical`	$a, c := \mathtt{T}, \neg b$	$b, c := \mathtt{T}, \mathtt{F}$
check	$\Box\neg(pc_1 = \mathtt{critical} \wedge pc_2 = \mathtt{critical})$	

Computing Abstraction [Saïdi/S, CAV'99]

Let concrete state have integer variables x and y.

Abstraction predicates given by γ: $\gamma(a)$ is $x > 0$ and $\gamma(b)$ is $y > 0$.

If the concrete formula C be $x = y$, the over-approximate abstraction of C is

D_i	$\vdash^? C \supset \gamma(D_i)$
a	$\not\vdash x = y \supset x > 0$
$\neg a$	$\not\vdash x = y \supset x \not> 0$
b	$\not\vdash x = y \supset y > 0$
$\neg b$	$\not\vdash x = y \supset y \not> 0$
$a \vee b$	$\not\vdash x = y \supset x > 0 \vee y > 0$
$a \vee \neg b$	$\vdash x = y \supset x > 0 \vee y \not> 0$
$\neg a \vee b$	$\vdash x = y \supset x \not> 0 \vee y > 0$
$\neg a \vee \neg b$	$\not\vdash x = y \supset x \not> 0 \vee y \not> 0$

Abstraction Example

Now suppose the concrete formula is $x > 1$.

The overapproximation is computed as

D_i	$\vdash^? C \supset \gamma(D_i)$
a	$\vdash x > 1 \supset x > 0$
$\neg a$	Not Relevant
b	$\not\vdash x > 1 \supset y > 0$
$\neg b$	$\not\vdash x > 1 \supset y \not> 0$
$a \vee b$	NR
$a \vee \neg b$	NR
$\neg a \vee b$	NR
$\neg a \vee \neg b$	NR

Efficient pruning of search space.

The QPQ Open Source Repository for Deductive Software Components

Deductive software is a scientific artifact that deserves publication, distribution, and archiving.

QPQ (see `www.qedproquo.org`) is a repository for peer-reviewed publication, dissemination, and maintenance of source code for deductive components.

Initial setup has been done, but we need volunteers for customizing the repository for deductive software.

QPQ will appear some time this year at `www.qpq.org`.

Conclusions

Need to make a careful distinction between macrologic and micrologic.

Higher-order logic is a powerful medium for embedding formalisms and integrating deductive procedures.

Contrast the encoding of modal logics in first-order logic (no modal operators) with that in higher-order logic.

There are no tradeoffs, only synergies.