PART 1: Historical context (partial)

PART 2: A Grand Challenge

PART 3: Combining computation and deduction in HOL-4
PART 1: Some history

A long time ago ...

► Robert Kowalski, *Predicate Logic as Programming Language*, 1973

“Our thesis is that predicate logic is a useful and practical, high-level, non-deterministic programming language with sound theoretical foundations.”

► P.J. Hayes, *Computation and Deduction*, 1973

“I will argue that the usual sharp distinction which is made between the processes of computation and deduction, is misleading. An interpreter for a programming language, and a theorem-proving program for a logical language, are structurally indistinguishable. Important benefits, both practical and theoretical, are obtained by combining the best of both methodologies.”
Bob Kowalski: deduction by a theorem prover as computation

Program:

\[ \text{DivOf}(x, y, z) = \begin{cases} 0 & \text{if } x < y \\ \exists w. \text{DivOf}(x-y, y, w) \land z = w+1 & \text{else} \end{cases} \]

Computation:

\[ \vdash \text{DivOf}(7, 2, z) = \exists w_1. \text{DivOf}(5, 2, w_1) \land z = w_1+1 \]
\[ = \exists w_1. (\exists w_2. \text{DivOf}(3, 2, w_2) \land w_1 = w_2+1) \land z = w_1+1 \]
\[ = \exists w_1. (\exists w_2. (\exists w_3. \text{DivOf}(1, 2, w_3) \land w_2 = w_3+1) \land w_1 = w_2+1) \land z = w_1+1 \]
\[ = \exists w_1. w_2. w_3. w_3 = 0 \land w_2 = w_3+1 \land w_1 = w_2+1 \land z = w_1+1 \]
\[ = (z=3) \]

Logic Programming

- Kowalski develops relational vision of programming as deduction
- execution by a resolution theorem prover
- Colmerauer develops Prolog
Pat Hayes: deduction by a theorem prover as computation

Program:

\[
Div(x, y) = \text{if } x < y \text{ then } 0 \text{ else } 1 + f(x - y, y)
\]

Computation:

\[
\begin{align*}
\therefore Div(7, 2) &= 1 + Div(5, 2) \\
&= 1 + (1 + Div(3, 2)) \\
&= 1 + (1 + (1 + Div(1, 2))) \\
&= 1 + 1 + 1 + 0 \\
&= 3
\end{align*}
\]

Functional Programming

- Hayes develops a functional vision of computation as deduction
- execution by resolution and paramodulation
- rewriting-based first-order languages (OBJ, Maude, ASF+SDF)
Moszkowski: executing temporal logic

- Ben Moszkowski, *Executing Temporal Logic Programs*, 1986

  “... temporal logic has been thought of as a tool for specifying and proving properties of programs ... This distinction between temporal logic and programming languages has troubled us since it has meant that we must simultaneously use two separate notations. ... One way to bridge the gap between logic and programs is by finding ways of using temporal logic itself as a tool for programming ...”

- Moszkowski’s *Tempura* system (1980s):
  - executes first-order temporal logic specifications
  - IMHO ahead of its time
Executing temporal logic – basic idea

- Division program (divides $X$ by $Y$ by repeated subtraction)
  \[\text{halt}(X < Y) \land \text{next}(X) = X - Y \land \text{next}(Y) = Y \land \text{next}(Z) = Z+1\]

- Execution as deduction of a normal form
  \[
  (X = 7 \land Y = 2 \land Z = 0) \\
  \land \bigcirc (X = 5 \land Y = 2 \land Z = 1) \\
  \land \bigcirc (\bigcirc (X = 3 \land Y = 2 \land Z = 2)) \\
  \land \bigcirc (\bigcirc (\bigcirc (X = 1 \land Y = 2 \land Z = 3)))
  \]

- **WARNING:** details here partial and oversimplified to fit on one slide!
Other kinds of computation besides execution

- Examples so far are using logic as a programming language
  - program is a term or formula specifying a problem
  - efficient deduction finds an answer
    (i.e. variable binding or value)

- Another kind of computation is checking the truth of a property
  - logic specifies a property of a system
  - efficient deduction determines if property true
    (or finds counterexample)

- Constructive logic is yet another way of linking deduction and computation
Model checking, theorem proving and computation

- Model checking consists of proving theorems about computation
  - “all computations of model satisfy property”
  \[ \forall c \in \text{computations}(\text{model}). \ \text{property}(c) \]

- Recent methods hybridise model checking and computation:
  - 0-In Search:
    * compute to an error state, then
    * explore all states around error state using model checking methods
  - Symbolic Trajectory Evaluation (STE, Bryant/Seger):
    * symbolically execute, then prove a Hoare logic like theorem:
      \[ \forall c \in \text{computations}(\text{model}). \ \text{pre}(\text{initial } c) \Rightarrow \text{post}(\text{final } c) \]

- Traditionally performed by a monolithic algorithm (BDD, SAT etc.)
Constructive logic

- Constructive proof of $\exists f. P(f)$ is computing $f$ satisfying $P$.

- Proof of

  $\exists f. \forall x y. f(x, y) \times y \leq x \land x - f(x, y) \times y < y$

  is an existential witness, essentially a functional program like:

  $\text{rec } f. \forall x y. f(x, y) = \text{if } x < y \text{ then } 0 \text{ else } 1 + f(x-y, y)$

- Example systems: Nuprl, Coq, Lego, Alf etc.

- Programming and hardware synthesis as (or side effect of) doing proofs

- Constructive proof as hardware/software engineering method not proven!
Computation = Logic + Control

- Formulas executed with a fixed interpreter/prover
  - programming achieved by writing formulas suitable for execution
    - control of execution implicit in form of formula
    - efficient formulas/programs might lose declarative clarity
  - compute truthvalue or counterexample by model checking

- Explicit control separated from declarative content
  - Milner’s LCF (1974): deduction/execution programmed in ML
  - various Prolog implementations with scheduling pragmas
Kowalski and Hayes propose convergence of computation and deduction
- Colmerauer partially implements the idea in Prolog
- functional programming viewed as efficient rewriting

Moszkowski’s makes temporal logic into a programming language

Model checking a special case of automatic theorem proving
- proves theorems about computations
- implemented using efficient computation engines

Constructive logic identifies proof objects as programs

Conclusion: diverse tangle of links between computation and deduction
Other work linking computation and deduction

Not an exhaustive list:

- Boyer/Moore: executable first-order theory of Lisp
- Executable specifications (VDM-SL, PVS)
- New relational and logic programming languages (Oz, Gödel, Mercury)
- Program refinement and formal synthesis
PART 2: A Grand Challenge

- Elegantly unify:
  - declarative specification
  - efficient computation
  - automatic formal checking
  - user-guided theorem proving

- This challenge has both theoretical/conceptual and practical aspects
  - theoretical/conceptual: build on Kowalski/Hayes/Moskowski vision
  - practical: link Prolog, rewriting, model checking, theorem proving . . .

- Submit other Grand Challenges by Oct 22
  
  http://umbriel.dcs.gla.ac.uk/NeSC/general/esi/events/Grand_Challenges/
PART 3: HOL and the Grand Challenge

- HOL evolving to a platform for computation and deduction (HOL-4)
- Separates object and meta language (following LCF)
- Emphasizes functional programming paradigm
- Only weak support for Prolog style programming
- Integrates external computation (BDD, SAT) with deduction
- **HOL only addresses a fragment of grand challenge**
Rest of the talk

- Generalisation of LCF approach
- Styles of linking external computation to theorem proving
- LCF-style model checking
- Cool examples (IMHO)
Review of Milner’s LCF approach

► Abstract type with a fixed kernel of operations
  - suppose abstract type consists of \( a : \text{thing} \) and \( f : \text{thing} \rightarrow \text{thing} \)
  - type-checking ensures only things of type \( \text{thing} \) are \( f(f(\cdots f(a)\cdots)) \)

► Milner’s key idea:
  - axioms and rules of logic are an abstract type
  - axioms have type \( \text{thm} \), rules have type \( \text{thm} \rightarrow \text{thm} \)
  - all values of type \( \text{thm} \) are constructed from axioms via rules

► Several systems use the ‘LCF approach’
  - HOL, Isabelle, Nuprl, Coq etc

► Key feature: program freely and type-checking takes care of soundness

Mike Gordon

University of Cambridge
The convergence of deduction and computation

Generalised LCF approach

- Code and data are logical terms or formulas
- Standard logic + additional judgements
- Standard rules of inference + judgement rules
- Execution is by application of rules of inference
  - standard rules: traditional inference programmed in metalanguage
  - additional judgement rules: external engines (coded in C, Java etc)
- Results are theorems in extended logic
- Idea evolved during PROSPER Esprit project
Examples of external engines in HOL-4

- SAT solvers like SATA, GRASP and CHAFF
  - efficiently determine if boolean formulae are satisfiable
  - compute models of satisfiable formulae

- Binary Decision Diagram (BDD) package
  - use representation judgements (see later)
  - judgement rules implemented with a BDD engine
SAT and BDDs

- **SAT**: algorithms for finding if a boolean formula is satisfiable
  - exist lots of competing algorithms
  - shallow analysis of huge terms
    (Alpha example from Tim Leonard: 10 MB, 1 million variables)
  - find easily reached bugs (few iterations of machine)

- **BDD**: data structure to compactly represent boolean formulae
  - many uses: model checking, symbolic simulation
  - deep analysis of small terms
  - find bugs after many iterations
**HOL-4’s HolSatLib and HolBddLib**

- HolSatLib and HolBddLib are HOL-4 libraries that use external oracles
  - create tagged theorems

- HolSatLib interfaces SAT solvers
  - currently SATO, GRASP and ZCHAFF supported
  - easy to add other solvers, naive file-based interface
  - provides coarse grain interaction

- HolBddLib is an interface to a BDD package
  - only BuDDy supported (Moscow ML + BuDDy = MuDDy)
  - ML and BuDDy GCs linked
  - fine grain interaction
HolSatLib

- **Naive file interface**
  - start with a term
  - print it to a file
  - invoke SAT solver on file to create a results file
  - parse results file and import results as a term

- **Can either check results inside HOL by proof** (*satProve*)

- **Or trust oracle and create tagged theorems** (*satOracle*)
**HolBddLib: a finer grain oracle**

- With HolSatLib one can only invoke SAT to
  - tell if a term is satisfiable or contradictory
  - find a model when satisfiable

- With HolBddLib one can link terms to BDDs
  - can test satisfiability and find models (like SAT)
  - but can also manipulate using BDD algorithms – e.g. to simplify
HolBddLib

**Goals**
- state-of-the-art BDD oracle
- clean interface consistent with the ‘LCF approach’

**Main idea is ‘BDD representation judgements’**

\[ \{t_1, \ldots, t_n\} \rho \ t \leftrightarrow b \]

- \(t_1, \ldots, t_n\) are assumptions (boolean terms)
- \(\rho\) maps logical variables to BDD variables (numbers)
- \(t\) is the term being represented
- \(b\) is the BDD representing \(t\)
  - under assumptions \(t_1, \ldots, t_n\)
  - with variable \(v\) in \(t\) corresponding to \(\rho(v)\) in \(b\)
Finite set of axioms and rules for deducing representation judgements
  next slide will have examples

Protected type \texttt{term\_bdd} of judgements
  analogous to protected type \texttt{thm} of theorems

Can program derived rules to perform complex BDD operations
  e.g. model checking
Rules that combine logic and BDD judgements

- Logically equivalent terms have the same BDD
  \[
  \frac{a_1 \vdash t_1 = t_2 \quad a_2 \rho t_1 \leftrightarrow b}{a_1 \cup a_2 \rho t_2 \leftrightarrow b}
  \]
  - \(t_2\) need not be a QBF, but we can infer a BDD for it
  - assumptions of theorems can be transferred to judgements

- Following rule links representation judgements to theorems
  \[
  a \rho t \leftrightarrow \text{TRUE}
  \]
  \[
  \frac{}{[\text{oracles: HolBdd}] \ a \vdash t}
  \]
  - a BDD-powered tautology checker

- Model checking as a derived rule
  \[
  a \rho "\text{property holds of model}" \leftrightarrow b
  \]
PuzzleTool

- Illustrates building a special purpose tool
  - meant to be analogous to ‘point tools’
- Combination of theorem proving and BDD calculation
  - use ‘LCF approach’ to using BDDs
- Solves a class of puzzles generalising ‘peg solitaire’
  - a ‘point tool’ for puzzle designers!
- Details in TPHOLs2002 paper
A ‘point tool’ for generalized solitaire

- A puzzle is defined by
  - a board
  - a set of moves
  - an initial state
  - a final state

- Tool will
  - find a solution to any puzzle, if one exists
    - solution is a sequence of transitions from initial to final state (a trace)
  - report if no solution exists and return set of reachable end states
    - an end state is one with no successors
    - can then pick a reachable terminal state to make a solvable puzzle
PuzzleTool as an inference rule

Derived rules for puzzle designers

\[
\vdash \text{Board} = \{\cdots\} \quad \vdash \text{Moves} = \{\cdots\} \quad \vdash \text{Init} = \{\cdots\}
\]

\[
\vdash \text{ReachableEndStates(Board, Moves, Init)} = \{\cdots\}
\]

\[
\vdash \text{Board} = \{\cdots\} \quad \vdash \text{Moves} = \{\cdots\} \quad \vdash \text{Init} = \{\cdots\}
\]

\[
\vdash \text{SolutionTrace(Board, Moves, Init)} = [\cdots]
\]

Looks like logic rules, but have BDD Inside
Business idea – the “killer app”?
Another example: Missionaries and Cannibals problem

- MCP is a standard AI toy problem
  - originally: 3 missionaries, 3 cannibals, boat of capacity 2
  - generalisation: \( n \) missionaries, \( n \) cannibals, boat of capacity \( k \)

- Solved using HOL-4 (LMS JCM, (5) 56-76, Aug 2002)
  - theorem proving: solvable if \( k \geq 4 \)
  - theorem proving: unsolvable if \( n \geq 2k \)
  - model checking: solve cases \( \{(n, k) \mid k < 4 \land n < 6\} \)

- Combines theorem proving and model checking

- Generated programs
  - instructions for ferrying people
  - generated as model checking ‘counterexamples’
Conclusions

- Convergence of deduction and computation not new
  - old ideas: logic and functional programming
  - now new provers, new logics, model checkers

- Grand Challenge
  - realise the dream of Kowalski and Hayes
  - still more concepts and engineering needed
  - cycles to spare (3GHz etc) \(\implies\) convergence now achievable?

- HOL’s contribution: unify theorem proving and model checking
  - link logic terms to efficient data-structures via a judgement calculus
  - intimately mix computation and deduction in single environment
Conclusions

- Convergence of deduction and computation not new
  - old ideas: logic and functional programming
  - now new provers, new logics, model checkers

- Grand Challenge
  - realise the dream of Kowalski and Hayes
  - still more concepts and engineering needed
  - cycles to spare (3GHz etc) $\implies$ convergence now achievable?

- HOL's contribution: unify theorem proving and model checking
  - link logic terms to efficient data-structures via a judgement calculus
  - intimately mix computation and deduction in single environment

THE END