Proceedings of the

Third HOL Users Meeting

Aarhus University, 1–2 October 1990

DAIMI PB – 340
December 1990
Foreword

This is a collection of papers, notes and copies of transparencies representing the talks and discussions of the Third HOL Users Meeting held at The Computer Science Department, Aarhus University 1 - 2 October 1990.

Glynn Winskel

Contents

- Programme.
- Abstracts of talks.
- Copies of transparencies and relevant papers.
THIRD INTERNATIONAL HOL USERS MEETING
Aarhus University, Denmark

PROGRAMME

DAY 1 (Tuesday, October 1)

09.00-10.00  REGISTRATION

10.00-10.15  Welcome: Glynn Winskel, Aarhus University, DK.

10.15-10.45  Mike Gordon, Cambridge University, UK.
              Current Research using HOL at Cambridge.  1 – 6

10.45-11.15  Tom Melham, Cambridge University, UK.
              A short overview of Version 1.12 of the system.  7 – 20

11.15-11.30  BREAK

11.30-12.00  Konrad Slind, University of Calgary, Canada.
              Details of a new HOL implementation in SML.  21 – 34

12.00-12.30  R.D. Arthan, ICL, UK.
              A High-Assurance Implementation of HOL.  35 – 46

12.30-14.00  LUNCH BREAK

14.00-15.00  Elsa Gunter, AT & T Bell Labs, USA.
              The Implementation and Use of Abstract Theories in HOL.

15.00-15.30  Myla M. Archer, University of California, USA.
              A Tree-editor Interface to HOL.  65 – 98

15.30-16.00  BREAK

16.00-16.30  Aarhus students.
              A Mutually Recursive Types Package in HOL.  99 – 114

16.30-17.00  Malcolm Newey, The Australian National University,
              Australia.
              Another Iteration in Arithmetic.  115 – 122

17.00  Discussion on future developments of HOL.

19.00  CONFERENCE DINNER (Jacob’s Barbeque)
<table>
<thead>
<tr>
<th>Time</th>
<th>Speaker(s)</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>09.30-10.30</td>
<td>Tom Melham, Cambridge University, UK.</td>
<td>A Mechanized Theory of the Pi-calculus in HOL.</td>
<td>123-150</td>
</tr>
<tr>
<td>10.30-11.00</td>
<td>BREAK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11.00-11.30</td>
<td>Flemming Andersen, TFL (A Telecomms Research Laboratory), DK</td>
<td>A Definitional Theory of UNITY in HOL.</td>
<td>151-162</td>
</tr>
<tr>
<td>11.30-12.00</td>
<td>Kim Dam Petersen, TFL (A Telecomms Research Laboratory), DK.</td>
<td>Construction of the P(omega) lambda calculus model in HOL.</td>
<td>163-174</td>
</tr>
<tr>
<td>12.00-12.30</td>
<td>Wim Ploegenaerts, IMEC, Belgium.</td>
<td>HOL theory for finite word length arithmetic.</td>
<td>175-188</td>
</tr>
<tr>
<td>12.30-14.00</td>
<td>LUNCH BREAK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.00-15.00</td>
<td>Jeff Joyce, University of British Columbia, Canada.</td>
<td>More Reasons Why Higher-Order Logic is a Good Formalism for Specifying.</td>
<td>189-232</td>
</tr>
<tr>
<td>15.00-15.30</td>
<td>Aarhus students.</td>
<td>A low-level circuit model in HOL.</td>
<td>233-252</td>
</tr>
<tr>
<td>15.30-16.00</td>
<td>BREAK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16.00-16.30</td>
<td>Cátia Marcondes Angelo, IMEC, Belgium.</td>
<td>The Verification of a Parameterized Mead and Conway Alu Core in HOL.</td>
<td>253-282</td>
</tr>
<tr>
<td>16.30-17.00</td>
<td>Richard J. Boulton, Cambridge University, UK.</td>
<td>The HOL Verification of ELLA Designs.</td>
<td>283-296</td>
</tr>
<tr>
<td>17.00-17.30</td>
<td>Ton Kalker, Philips Research Laboratories, The Netherlands.</td>
<td>HOL Semantics of SILAGE.</td>
<td>297-314</td>
</tr>
</tbody>
</table>
AUTHOR: M.J.C. Gordon

TITLE: Current Research using HOL at Cambridge

SHORT ABSTRACT:
Discussion of some activities with HOL since the last meeting.

AUTHOR: Tom Melham

TITLE: A short overview of Version 1.12 of the system.

AUTHOR: Konrad Slind

TITLE: Details of a new HOL implementation in SML.

AUTHOR: R.D. Arthan

TITLE: A High-Assurance Implementation of HOL

SHORT ABSTRACT:
ICL are engaged in an IED sponsored project to develop a highly assured version of HOL for use in industrial applications. The paper will report on this work.

AUTHOR: Elsa L. Gunter

TITLE: The Implementation and Use of Abstract Theories in HOL

SHORT ABSTRACT:
At the last HOL meeting, I proposed that a notion of abstract theories be added to HOL. This year, I will discuss an implementation of that proposal which was implemented mostly on top of the existing notion of theories in HOL. We will discuss what modification were made to the underlying system, why they were made, why it is sound to have made them, and in what sense are they upwards-compatible. We will discuss how abstract theories are used in a variety of setting. Finally, time allowing we will discuss some of
the ways in which this implementation falls short of giving the user the full functionality that one might expect from abstract theories and what it would take to overcome these shortcomings.

AUTHOR: Myla M. Archer

TITLE: A Tree-editor Interface to HOL

SHORT ABSTRACT:

We have developed a tree-editor interface (PM) to HOL, based on an Emacs-based programmable tree editor (Treemacs) developed at the Univ. of Illinois. PM ("proofmanager") permits convenient access to subgoals and convenient proof display, and maintains proof soundness. There are additional useful features, and plans for extending the tool.

AUTHOR: Malcolm Newey

TITLE: Another Iteration in Arithmetic.

SHORT ABSTRACT:

I have used Tom Melham’s axiomatisation of natural numbers (the one appearing in HOL88 1.11) but have changed the theory so much that I will (no doubt) incur the wrath of many HOL users when I suggest we make a change to this important standard fragment of the Basic HOL System. Nevertheless, I do intend to argue that the benefits of a major change in this area, even at this late stage, are worth the inconvenience that will be caused to users with a substantial investment in proofs using the current package.

AUTHOR: Tom Melham

TITLE: A Mechanized Theory of the pi-calculus in HOL.

SHORT ABSTRACT:

This talk will describe work in progress on the construction of a mechanized formal theory in HOL of the pi-calculus, a process algebra developed by Milner and others for modelling communicating systems in which processes can have changing structure. The aim is to support reasoning in the pi-calculus about applications as well as proofs about the pi-calculus itself.
AUTHOR: Flemming Andersen

TITLE: A Definitional Theory of UNITY in HOL

SHORT ABSTRACT:

Higher Order Logic is powerful enough to model the axiomatic definition of the UNITY logic as defined in [Chandy, Misra: Parallel Program Design, A Foundation. Addison Wesley 1988].

A method for defining relations in HOL using fixpoint equations is presented. It is shown that the UNITY relation ‘leadsto’, which models the graph of a transitive and disjunctive closure of binary relations, is a special case of these relations.

AUTHOR: Kim Dam Petersen

TITLE: Construction of the P ω lambda calculus model in HOL

SHORT ABSTRACT:

The P omega model may be constructed in Higher Order Logic, using the description given in [Barendregt: The Lambda Calculus, North-Holland 1984].

The construction of the model in HOL is presented, and an outline of solving selected reflexive domain equations using the model is given.

The final goal is to make a type package that is able to solve general reflexive domain equations.

AUTHOR: Wim Ploegaerts

TITLE: “HOL theory for finite word length arithmetic”

SHORT ABSTRACT:

The ultimate goal of the ongoing research in our group is the verification of a silicon compiler for DSP (Digital Signal Processing) applications. As a first step, it has been tried to build the libraries that are required for the definition of the behavior of hardware components (such as adders, multipliers, ...) in terms of integer numbers. The work includes e.g. an extention of the library “zet” (definition of integer division and remainder, a normalization conversion for integer expressions and some related tactics) and the
definition of the denotation and representation functions for both signed and unsigned arithmetic.

AUTHOR: Jeff Joyce

TITLE: More Reasons Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware

SHORT ABSTRACT:

The HOL community is often challenged to justify the choice of higher-order logic as a specification language over conventional description languages such as VHDL or less expressive formalisms such as first-order logic. The question, “why higher-order logic?” was partly answered by Gordon’s 1985 paper, “Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware”. Our (proposed) presentation will discuss why higher-order logic, in particular (as opposed to less expressive formalisms such as first-order logic), is a very good formalism for specifying and verifying hardware. We focus on two main reasons: (1) the ability to support generic specifications of hardware, (2) the ability to embed special-purpose formalisms such as temporal logic.

AUTHOR: Catia Marcondes Angelo

TITLE: “The Verification of a Parameterized Mead & Conway Alu Core in HOL”

SHORT ABSTRACT:

This work is concerned with the formal verification of the parameterized Mead & Conway Alu module core implemented in the Cathedral II Silicon Compiler using the Proof Assistant HOL from Cambridge University. The correctness proof is divided in two parts: the structural and the behavioral reasoning. The structural verification consists of a formal alu decomposition reflecting the basic design process knowledge to prove the hardware implementation behaves like an ideal alu. The behavioral reasoning formally verifies the ideal alu meets the specification.
AUTHOR: Richard J. Boulton

TITLE: The HOL Verification of ELLA Designs

SHORT ABSTRACT:

ELLA is a hardware design language developed at the Royal Signals and Radar Establishment (RSRE) and marketed by Praxis. It supports simulation models at a variety of different abstraction levels.

A preliminary methodology for reasoning about ELLA designs using HOL is described. Our approach is to semantically embed a subset of the ELLA language in higher order logic, and then to make this embedding convenient to use with parsers and pretty-printers. There are a number of semantic issues that may impact on the ease of verification. We discuss some of these briefly. We also give a simple example to illustrate the methodology.

---

AUTHOR: Ton Kalker

TITLE: HOL Semantics of SILAGE

SHORT ABSTRACT:

SILAGE is the input language for a prototype silicon compiler for DSP applications. An exercise to endow SILAGE with a formal semantics in HOL logic revealed several weak points of SILAGE. In the presentation I will argue that designing a language like SILAGE will benefit from using mathematical methods.
Developments of HOL

- HOL98.1.12 to be released in November (details in talk by Tom Melham)
- New edition of documentation in progress
  - REFERENCE to be completed
  - DESCRIPTION and TUTORIAL revised and extended
  - Quick reference cards in preparation
  - Compatible with Calgary HOL
  - Slides for HOL courses to be distributed with the system
- X-windows based demo tool (John Van Tassel) and theorem retrieval system (Richard Boulton) available

Kinds of Project

- Small projects
  - Typically a PhD student or individual researcher
  - Not supported by an external grant
- Large Projects
  - Several researchers, at least one full-time
  - Some external support

Small Projects

- Protocol verification
- Architecture specification
- \( \pi \)-calculus
- Compositional proof systems
- Formalisation of real-time systems
- VHDL semantics
Large Projects

- I/O hardware verification
- Viper UART & VISTA translator verification
- Proof analysis and accounts
- Verified proof-checker
- SAFEMOS
- Processor verification
- CHEOPS
- HOL verification of ELLA designs

Computer Architecture Specification

- Research Goal:
  Augment informal architecture specs with formal ones to reduce problems of ambiguous spec meaning, and to allow the use of theorem provers for testing those meanings

- Research Methods:
  To formalise industrial-style specs, express them in HOL, and prove properties of them

- Status:
  Going for 1.5 years with some simple examples completed Annotated bibliography on architecture specification has been produced

- Researcher: Tim Leonard — PhD student, VAX architect with DEC

Protocol Verification

- Research Goal:
  Verification of code implementing protocols

- Research Methods:
  Specifications and real-time programs are modelled in HOL

- Status:
  In progress for 2 years with a preliminary case study available as part of the HOL documentation

- Researcher:
  Rachel Cardell-Oliver — PhD Student at Cambridge, Cadet in Australian DSTO

π-Calculus in HOL

- Research Goals:
  Provide proof support for the π-calculus (higher-order successor to Milner's CCS) in HOL, check proofs of existing meta-theorems, and investigate the calculus for modelling systems

- Research Methods:
  Define the calculus syntax as a recursive concrete type and specify its semantics via structured operational semantics. Existing informal proofs will then be replicated in HOL

- Status: Just started (syntax defined in HOL, some simple lemmas proved)

- Researchers:
  Tom Meiham and Mike Gordon
Compositional Proof Systems

- Research Goals:
  To build tools to support inductive definitions of compositional proof systems and automatic generation of induction tactics

- Research Methodology:
  Driven by the need to define structured operational semantics based on minimal relations. The \( \pi \)-calculus and other process algebras will be used as examples

- Status:
  Some examples already done interactively. Automatic tools about to be written

- Researchers:
  Tom Melham and Juanito Camilleri

Formalisation Of Real-Time Systems

- Research Goals:
  Define a small language containing explicit timing constructs as defined by a deterministic scheduler. This environment should be seen as implementing some of the ideas from the world of process algebras, but with explicit real-time characteristics

- Research Methods:
  Specify a scheduler in HOL and verify its safety and liveness properties. An evaluation of the constructs based on this approach will be done along with a final definition of the syntax and a formalisation of the semantics

- Status:
  Early days

- Researcher:
  Neil Viljoen - PhD student

VHDL Semantics

- Research Goal:
  To define a subset of VHDL tractable for formal methods, to specify its semantics in HOL and hence to develop a set of proof tools

- Research Methods:
  Build on the results and lessons of the HOL-ELLA project with the understanding that VHDL is a much more complex language

- Status:
  To start on 1 October 1990

- Researcher:
  John Van Tassel — PhD Student

I/O Hardware Verification

- Research Goal:
  To design and verify a UART whose initial target is the VIPER, and will serve as a prototype of Transputer link verification (SAFEMOS)

- Research Methods:
  Specification and verification to be as general as possible so that the proofs can hopefully be reused. Fabrication will take place with XILINX user-programmable gate arrays

- Status:
  Design and verification complete; fabrication to be conducted as part of the HOL-ELLA project

- Researcher:
  John Herbert
VISTA Translator Verification

- **Research Goal:**
  Define the semantics of VISTA in HOL. Define and verify a translator from VISTA to VIPER machine code written in the HOL logic.

- **Research Method:**
  Define the syntax and a direct denotational semantics of both source and target language in HOL with the translator as a primitive recursive function.

- **Funding**
  MOD (RSRE)

- **Researcher:**
  Paul Curzon

---

Proof Certification

- **Research Goal:**
  Understand the structure and parameterisation of large proofs; develop and test a technology for producing 'proof deliverables' (idea proposed by Newey, Hanna).

- **Research Method:**
  - Modify HOL to produce proof texts
  - Code a checker for them in a programming language supported by HOL
  - Verify the checker

- **Status:**
  SERC/MOD funding promised. Target start date is 1.4.91

- **Researcher:**
  Avra Cohn

---

Proof Analysis

- **Research Goal:**
  Better understanding of the relationship between proofs and tactics. Implementation of prototype tools to generate proof summaries from tactics.

- **Research Methods:**
  Detailed study of "naturally occurring" proofs and replicating them in HOL. Experimental programs to generate proof narratives from tactics have been written.

- **Status:**
  Funded by SERC

- **Researcher:**
  Avra Cohn

---

SAFEMOS I

- **Research Goal:**
  - Demonstrate the possibility of totally verified systems by creating a verified application program running on a verified processor by means of a verified compiler
  - Develop a methodology for building verified real-time systems

- **Research Method:**
  Example driven with a programming language based on a subset of OCCAM, and the processor based on the Transputer. The proof will build on existing work such as the CLINC stack (Austin, Texas) and Joyce’s work on Tamarack (Cambridge).
SAFEMOS II

- Partners:
  INMOS, SRI International Cambridge Research Centre, Oxford Programming Research Group, Cambridge Computer Laboratory

- Funding
  SERC/DTI (IED)

- Status:
  Started 1.1.90

- Researchers:
  - INMOS: David Shepard + 2
  - SRI: Roger Hale, \( \frac{1}{2} \) John Herbert
  - PRG: Jonathan Bowen, Paritosh Pandya
  - CL: Mike Gordon, Juanito Camilleri

HOL–ELLA I

- Research Goal:
  - Specification of the formal semantics of a subset of the ELLA hardware description language in HOL with theorem-proving support for reasoning about ELLA designs
  - Development of a methodology for adding formal methods to conventional CAD

- Research Method:
  Translate ELLA to HOL via explicit semantics and test the methodology by designing XILINX chips

HOL–ELLA II

- Funding:
  Initially SERC/DTI project with Praxis. Now a pure SERC project

- Status:
  Running since 1.10.89, with a simple case study completed. Subset of ELLA chosen and HOL semantics defined and implemented

- Researchers:
  Richard Boulton (until 30.9.90), John Harrison (from 1.10.90), \( \frac{1}{2} \) John Herbert

Processor Verification

- Research Goal:
  To design and verify a processor for real-time control applications as part of the SAFEMOS and HOL-ELLA projects

- Research Methods:
  Develop standard techniques which will transfer to the verification of the main Transputer-like SAFEMOS processor

- Status:
  Early days

- Researcher:
  John Herbert
CHEOPS I

• Research Goal:
  - Interface HOL to CATHEDRAL for the verification of synthesis functions (IMEC)
  - Improve the HOL environment through new libraries, and a better user interface (Cambridge)

• Research Methods at Cambridge:
  General-purpose parsers and pretty-printers driven from declarative inputs. Interface tools implemented in X-windows.

• Partners:
  IMEC (Belgium), Philips ERL (Netherlands), Cambridge Computer Laboratory

CHEOPS II

• Funding:
  Esprit BRA

• Status:
  Running since 1.8.89 with the syntax tools complete. X-windows interface started with a prototype developed in an enhanced GNU emacs.

• Researchers:
  - IMEC: Luc Claesen, Wim Ploegaerts, Catia Angelo
  - Philips: Ton Kalker
  - Cambridge: John Van Tassel (until 30.9.90), Sara Kalvala, Andy Gordon (from 1.1.91)

Summary

• HOL88 system is now stable.

• New improved Standard ML implementations are almost ready (Viz. HOL90 from Calgary and ICL HOL).

• Polished documentation (for both HOL88 and HOL90) coming soon.

• Increased automation and improved interfaces are on the horizon.

• Many projects announced at first HOL Users Meeting are now up and running.

• Lots of new projects, many unrelated to hardware verification, are starting up.
• Summary of changes already made.
• Some proposed changes.

T F Melham, 28 September 1988
Current state of version 12

- Summary of changes already made:
  - preterms, and user-definable typechecking
  - revised ‘resolution’ tools
  - paired beta-conversion
  - new tactics, rules, etc.
  - revision/optimization of some tactics, rules, etc.
  - natural number MOD and DIV theorems added
  - total rewrite of type definition package
  - revised set theory library
  - new ‘contrib’ directory
  - many internal ML functions deleted/hidden
New ML type of raw terms

• A new ML type of untyped terms:
  
  preterm = var of string  
  | const of string  
  | comb of preterm # preterm  
  | abs of preterm # preterm  
  | typed of preterm # type  
  | antiquot of term

has been added, together with:

preterm_to_term : preterm -> term

which invokes the standard HOL typechecker.
User-definable typechecking

- To enable user-defined typechecking:
  
  set_flag('preterm',true);

- To install user-defined typechecking:

  let preterm_handler p = \(\text{function}\)

  where \(\text{function}\) has type:

  : preterm -> term

- Note: this has not been used much yet.
Revised ‘resolution’ tools

- In version 11, \texttt{RES\_CANON} does:

\[
\begin{align*}
\neg t & \Rightarrow t \Rightarrow F \quad \text{(at outermost level)} \\
t_1 \land t_2 & \Rightarrow t_1, t_2 \\
(t_1 \land t_2) \Rightarrow t & \Rightarrow t_1 \Rightarrow (t_2 \Rightarrow t) \\
(t_1 \land t_2) \Rightarrow t & \Rightarrow t_1 \Rightarrow t, t_2 \Rightarrow t \\
t_1 = t_2 & \Rightarrow t_1 = t_2, t_1 \Rightarrow t_2, t_2 \Rightarrow t_1
\end{align*}
\]

- In version 12, \texttt{RES\_CANON} also does:

\[
!x. t_1 \Rightarrow t_2 \Rightarrow t_1 \Rightarrow !x. t_2 \quad (x \text{ not free in } t_1)
\]

- For example:

\[
\vdash !m \land n \land p. m < n \land n < p \Rightarrow m < p
\]

in version 11 gets transformed to:

\[
\vdash !m \land n \land p. m < n \Rightarrow (n < p \Rightarrow m < p)
\]

but in version 12 it gets transformed to:

\[
\vdash !n \land m. m < n \Rightarrow (!p. n < p \Rightarrow m < p)
\]
Effect of this change

- Given the assumptions:
  \[ a < b \quad \text{and} \quad b < c \]

the tactic:

\[
\text{IMP\_RES\_TAC} \vdash !m\ n\ p.\ m \prec n \land n \prec p \Rightarrow m \prec p
\]

will produce the assumptions:

\[
[ !p. \ b \prec p \Rightarrow a \prec p ] \quad [ !p. \ c \prec p \Rightarrow b \prec p ]
\]

instead of freezing the variable \( p \):

\[
[ b \prec p \Rightarrow a \prec p ] \quad [ c \prec p \Rightarrow b \prec p ]
\]

- Consequence:
  
  - more new consequences generated, so
  
  - more assumptions to work with,
  
  - e.g. RES\_TAC will solve the goal more often;
  
  - \textbf{BUT:} your existing proofs may not work.
Paired beta-conversion

- New function for paired beta-conversion:

\[
\text{PAIRED\_BETA\_CONV} : \text{conv}
\]

- For example, given the term:

\[
"(\langle x_1, \ldots , x_n \rangle . t) (t_1, \ldots , t_n)"
\]

the new conversion proves that:

\[
\vdash (\langle x_1, \ldots , x_n \rangle . t) (t_1, \ldots , t_n) = t[t_1, \ldots , t_n/x_1, \ldots , x_n]
\]

- It also works on arbitrarily-nested tuples.

---

Related conversion

- LET - conv
  - coming soon
Natural number division

• In version 11:

|- 0 < n ==> (?r q. (k = (q * n) + r) \ r < n)
|- MOD k n = @r. ?q. (k = (q * n) + r) \ r < n
|- DIV k n = @q. (k = (q * n) + (k MOD n))

• In version 12:

|- 0 < n ==> (?r q. (k = (q * n) + r) \ r < n)
|- 0 < n ==> (k = ((k DIV n)*n)+(k MOD n)) \ (k MOD n) < n

• Plus built-in theorems, for example:

|- !k. k MOD (SUC 0) = 0
|- !n. 0 < n ==> (!k. (k DIV n) <= k)
|- !n r. r < n ==> (!q. ((q * n) + r) DIV n = q)
|- !n k. k < n ==> (k MOD n = k)
|- !n. 0 < n ==> (!k. (k * n) MOD n = 0)
|- !n. 0 < n ==> (0 MOD n = 0)

• Other arithmetic theorems have also been added.
Type definition package

- Has been totally rewritten.

- The code is now:
  - Cleaner and better organized,
  - faster (sometimes much faster),
  - much better documented.

- The main changes visible are:
  - modifications to low-level tools,
  - define_type now has a proper parser,
  - different variable names/priming behaviour.

- Coming soon:
  - more general function definitions
Revised set theory library

- Reorganization:

<table>
<thead>
<tr>
<th>Old name</th>
<th>New name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sets</td>
<td>finite_sets</td>
<td>finite sets</td>
</tr>
<tr>
<td>all_sets</td>
<td>sets</td>
<td>finite and infinite sets</td>
</tr>
<tr>
<td>set</td>
<td>pred_sets</td>
<td>predicates-as-sets</td>
</tr>
</tbody>
</table>

- The library sets now supports the syntax:

"\{x_1, \ldots, x_n\}\"
"\{x \mid P[x]\}\"
"\{t[x_1,\ldots,x_n] \mid P[x_1,\ldots,x_n]\}\"

- Proof support for this notation (for example):

```
SET_SPEC_CONV "t in \{x \mid P[x]\}\"
```

returns:

```
\|- t in \{x \mid P[x]\} = P[t/x]
```

- The parser/prettyprinter support is general.
The contrib directory

- **Contents:** user contributions

- **Purpose:**
  
  for easy distribution of items such as code, theories, and documentation which are useful but not suitable for the library

- **Current contents:**
  
  - CPO: complete partial orders (jac1)
  - PNF: prenex normal form (cj)
  - batch-hol-tool: simulate interactive session (rjb)
  - franz-cl-th: translate theories to common lisp (jac)
  - knuth-bendix: equational reasoning (slind)
  - select: how to deal with epsilon (grahamb)
  - temporal: temporal logic in HOL (jasuja)
  - tooltool: suntools interface (des)

- **Further contributions very welcome!**
Hiding/Deleting Internal Functions

- The goal is:

```
bound
ML id's
```

```
one-to-one

manua

entries
```

correspondence

- Many identifiers already deleted/hidden.

- For example:

<table>
<thead>
<tr>
<th>Redundant</th>
<th>Old versions</th>
<th>Unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND_CLAUSE1</td>
<td>HOL_IMP_RES THEN</td>
<td>form_class</td>
</tr>
<tr>
<td>AND_CLAUSE2</td>
<td>HOL_MATCH_MP</td>
<td>form_frees</td>
</tr>
<tr>
<td>AND_CLAUSE3</td>
<td>HOL_PART_MATCH</td>
<td>form_tyvars</td>
</tr>
<tr>
<td>AND_CLAUSE4</td>
<td>HOL_RESOLVE_THEN</td>
<td>form_vars</td>
</tr>
<tr>
<td>AND_CLAUSE5</td>
<td>HOL_RES_THEN</td>
<td>(etc)</td>
</tr>
<tr>
<td>(etc)</td>
<td>OLD_RES_TAC</td>
<td></td>
</tr>
<tr>
<td>(etc)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


Proposed changes/developments

- Future developments and proposed changes:
  - rationalize resolution
  - faster rewriting
  - automatic abstract type definitions
  - new type-definition primitive
  - library revision
  - new version of the manuals
An SML Implementation

of
hol88

Konrad Slind
+

Graham Birtwistle,
Rajagopala Nagarajan,
Todd Simpson

Funded by:
- NSERC Strategic Grant `88-`92
  Verifying Hardware Designs

- Equipment grants from
  NSERC and CMC
- Lisp + ML + ML compiler
- HOL (logic) implemented using PPLAMBDA code. Requires "bending" of HOL objects.
- Primitive routines for terms are intricately coded and deserve to be held to the light.
- Comments: sparse, multilingual, etc... and now for a tricky bit?
- Theory files have obscure, Lisp-dependent syntax
- Unclear as to how the system builds
- SML
hol 90

- "done" (v.1.11 \ contributed libraries)
- provides same interface
- \( \approx 15,000 \) lines of SML
- \( \approx 1.3 \) M. core
- Speed. Not emphasized.
  - faster in forward inference.
  - slower in parsing terms
  - very slow in reading from theory files.

- in PolyML

- Core Language only.
Which SML?

- all of them
- Implementation requires features not in the Standard:
  1. Embedding HOL in SML:
    - New top level with macro expansion phase

  typical top level:
  let top_level () =
    ( prompt ();
    print ( eval ( read () ));
    top_level () );
New top level:

```plaintext
fun top_level () =
  (prompt ();
   print (eval (read (read_HOL ())));
   top_level ());

let tm = "v \& v"
in
...
=>
let tm = mk_conj (mk_var ("v"), mk_type "bool"),
   mk_var ("v"), mk_type "bool"
in
...
```

PolyML allows the installation of user-defined top levels

MK-YACC
2. pretty printing terms, types, theorems. 

Given "final" top level:

```ml
fun top_level () :
  ( prompt () ;
    print + ( eval ( read ( read_HOL () ))) ;
    top_level () ) ;
```

POLYML allows the extension of the pretty printer of SML.

Why no structures:
- had used them up to a point in time, but
  1. gc costs, core size
  2. signature maintenance
  3. Flakiest part of implementation
- We plan to add them before releasing HOL.90
A tour of the implementation
- divides nearly between - ML/Lisp
  - domed/undoced

ML

Libraries (Sets, groups, processes, ...)
More powerful definitional mechanism
(Recursive types)

Basic theories (N, sum, one, ...)
Backwards proof interface (goal stack)
Conversions, tacticals, tactics

Derived rules of inference
ithm ADT (primitive rules of inference, axioms)

basic definitional mechanism

theory interface
  parsing HOL (quotation, antiquotation, top level)
type inference, pretty printing
Symbol table
terms
types
Archaeological Results

- hol88 uses variable sharing and conversion notes for fast rewriting
- "bootstrapping"
- we actually have an ADT of them only primes when it has to
- antiquote suppresses type inference:
  let v = "v:bool"
  in
  "\( \forall (v=1) \)" passes
  but
  "\((v:bool) \land (v=1)\)" fails
- ADT \texttt{thm} shouldn't use quotation, i.e.
  \[ \text{REFL} \; \texttt{tm} = \texttt{ab-thm} (\texttt{ES}, \texttt{"\texttt{tm}:\texttt{tm}\"}) \]
  for the correctness of \texttt{thm} depends on correctness of \texttt{read.HOL}.
- aconv is "dynamic deBruijn".

  \( \lambda x. (x:bool) \) " \( \lambda x. (x:*i) \) " \( \Rightarrow \) true

can we get inconsistency?

- hol88 uses property lists, we have a symbol table
  some of hol88 term handling routines
  are hardwired in, e.g. mk-conj
  inherited from LCF

- some cleanup, for example theory of
  pairs - we can do backwards proof
  of some theorems there, instead of
  using mk-thm.

- theory file interface cleaned up. Parser
  info not held as theorems.

- theory cache.

---
Speed Revisited

Completing the group

Sun3 1731 thms 2185 s. ho188
31 s. ho190/PolyML
16 s. ho190/NJ3ML
10 s. Sun4/NJ3ML

To do

- make parsers identical
- speed up parser,
  - theory file interface
  - conversion nets for rewriting
- move to modules system
- attempt to move to NJ3ML
- better DS/A for rewriting
- redesign theory interface
- hooks for UI work
ML → SML

let x = ... in let y = ... in

→
let val x = ...
val y = ...
in...
end

letrec or functional let → fun
let x = e → val x = e

;; → ;
[a; b; c] → [a, b, c]

[a,b] = [(a,b)] → [a,b] — beware

Cambridge allows leaving the else arm off conditional. Not SML
- # used for different purposes in SML

- \( (1, 2, 3) \equiv (1, (2, 3)) \rightarrow (1, 3) \) \hspace{1cm} \text{danger! semantic shift}

- Exceptions much more powerful in SML

- \& \rightarrow \text{andalso}
  \\| \rightarrow \text{orelse}

- \( b \Rightarrow e_1 \mid e_2 \rightarrow \text{doesn't exist} \)

- \( \lambda x. e \rightarrow f x \Rightarrow e \)
  \( \lambda x. e \rightarrow f x \Rightarrow f y \Rightarrow f z \Rightarrow e . \)
  \( f x \) is an SML keyword, so don't use \( f x \) for variable naming.

- \( \ldots \lambda \rightarrow (\ast \ldots \ast) \)

- Common problem:
  \( (\ast \text{ function } f \text{ has type } (\ast \rightarrow \ast) \rightarrow \text{bool} \)
  \( \ast \) taken as end of comment!
Cooperation Anyone?

- many interesting aspects of this work.

- If you want to dive in and add to the system, we would be happy.
A High Assurance Implementation of HOL

Rob Arthan
ICL Secure Systems
The FST Project

(DTI Supported Project IED/1563)

Cambridge University
- foundational research

ICL Secure Systems
- lead partner
- HOL technology and applications

Kent University
- VERITAS+

Program Validation Limited.
- SPARK
Background

- Business is high-assurance secure systems.
- HOL used in applications for about 4 years
- Proof deliverables produced on several real secure system developments
- Piecemeal work to tailor the system to our needs:
  - Tied off loop-holes (e.g. mk_thm)
  - Metalanguage and theory support (Z, hardware DA links, methods support)
Directions

Want to improve capability to produce formal proofs of critical properties in real applications.

HOL offers a tried and trusted basis.
We would like:

- Greater integrity (possibility of accidental or malicious proof of F)

- Better support for the languages our customers want (Z, Ada)

- Improved ease of use (proof work is currently very costly)
High Assurance HOL

Basic idea (i.e. long term goal):

(a) formally specify logic

(b) formally specify critical properties of the proof development system in terms of (a)

(c) verify implementation (or at least design against (b)).

Result is a proof development system with assurance level analogous to that of a US A1+ (= UK level 6) secure system.

Want to take sensible and useful steps towards this.
State of Play

- Have formal specifications of
  - language and inference rules
  - theory hierarchy
  - semantics
  (= chapters 9&10 of the manual)

- Have a prototype system in Standard ML
  - Logical primitives based on formal spec
  - Theorem proving superstructure based on Cambridge documentation and implementation
  - Basis for experiments with proof automation etc.
Current Work

- designing a system to be implemented to product standards

- rationalising logic spec and planning out integrity proof

- modest steps in proof automation

- support for $\mathbb{Z}$

- user interface issues (e.g. subgoal package improvements)

- compatibility and interchange
Logic Specification

- Specified HOL as a deductive system.
  - language
  - rules of inference
  - theories with theorems defined via derivability
- PDS as a machine with a theory hierarchy in its state, making "derivable" transitions on the theories in it

- Treatment is quite long (30 pages of theory listing without consistency proofs)
- Now looking at incorporating semantics and reorganising to ease proof task
TOP LEVEL SPEC OF THE PDS

HOL_STATE

parent : string→string→bool
theory : string→THEORY

(antisymmetric(ancestral parent)) ∧
(rooted parent) ∧
(∀init·root parent init ⇒ theory init = INIT)
(order_preserving theory parent extends)

HOL_SYSTEM ≜ (*INPUT × HOL_STATE) →
(HOL_STATE × *OUTPUT)

safe : (*INPUT, *OUTPUT) HOL_SYSTEM → bool

∀hol_system·safe hol_system ⇔
∀input hol_state·
consistent_state hol_state ⇒
consistent_state(FST(hol_state(input, hol_state)))

Similarly for conservative extensions etc.
Proof Automation

• Many useful modest steps to be made

• Incorporate useful techniques from research on other theorem provers

• Have prototyped:
  
  • Several tautology proving algorithms
  
  • Divide-and-conquer techniques for finding normal forms (e.g. polynomials over a commutative ring)
  
  • Tools for justifying various forms of definition

• Future work

•Decision procedures etc.

• Improved support for recursive types
User Interface

- Main concerns are "deep" interface issues rather than "surface" ones

- Want a consistent and comprehensive collection of tools within HOL.

- Have made some small experiments
  - Subgoal package using a theorem to represent the state (gives automatic validation of tactics safely, and other useful features).
  - Term surgery-style tools
  - Cross-referencing tools

- Future work
  - User interfaces for other languages (e.g. Z)
  - Better support for programming tactics and conversions
The Implementation and Use of Abstract Theories in HOL.

Elsa Gunter
Bell Labs
What do we want from abstract theories?

Abstract theory should create a context of assumed information, and give that information scope.

Within an abstract theory, attempts to prove theorems have access to that information.

Outside abstract theory, access to that information is removed. Theorems proved using it have had it abstracted out.

To use a theorem from within an abstract theory outside, must supply abstracted information.
What will having abstract theories do for us?

Allow for the development of general mathematical theories.

Allow for modularity and cleaner organization of proofs in "theorem proving in the large."

Could be used in organizing verification proofs to make the implementation specification layers clear.
More specifically, what do we want from abstract theories?

An abstract theory should take a signature consisting of
- list of type variables
- list of term variables
- list of assumptions

and create a scoped context in which, when proving theorems, the type variables and term variables are treated as type and term constants, and assumptions are treated as axioms.
A Definitional Theory of
UNITY
in
Higher Order Logic
Overview

- Programs as state transitions and their properties
- Defining program properties in Higher Order Logic
- The transitive & disjunctive closure of progress
- Introducing bounds on progress properties
- Looking for a generalised closure definition
- How UNITY is derived
- Conclusion
Programs as state transitions

Program Properties

Safety, $R_s$
Liveness (progress), $R_l$
Direct, inferred properties
Defining program properties in Higher Order Logic

**Safety**

\[
p \text{Rs} \quad q \equiv \\
\forall s, st \in \text{Prog} : \ R_{safe}(p, q, st, s) \\
(\text{satisfied for all possible states})
\]

**Chandy, Misra**: 

\[
p \text{unless} \quad q \equiv \\
\forall s, st \in \text{Prog} : \ (p s \land \neg q s) \Rightarrow \ (p (st s) \lor q (st s))
\]

**Liveness (progress)**

\[
p \text{Rp} \quad q \equiv \\
\forall s : \exists st \in \text{Prog} : \ R_{trans}(p, q, st, s) \\
(\text{express the existence of a state transition})
\]

**Chandy, Misra**: 

\[
p \text{ensures} \quad q \equiv \\
\forall s, st \in \text{Prog} : \ (p s \land \neg q s) \Rightarrow q (st s)
\]
The transitive and disjunctive closure of progress

What properties are wanted:

- \( P_1 \rightarrow_{T} P_3 \quad \Rightarrow \quad P_1 \rightarrow_{C} P_3 \)
- \( P_1 \rightarrow_{T} P_3 \land P_3 \rightarrow_{T} P_4 \quad \Rightarrow \quad P_1 \rightarrow_{C} P_4 \)
- \( P_1 \rightarrow_{T} P_3 \land P_2 \rightarrow_{T} P_3 \quad \Rightarrow \quad P_1 \lor P_2 \rightarrow_{C} P_3 \)

An axiomatic definition:

- \( \frac{P \rightarrow_{T} q}{P \rightarrow_{C} q} \), \( \frac{P \rightarrow_{C} q, q \rightarrow_{C} r}{P \rightarrow_{C} r} \), \( \frac{P_1 \rightarrow_{C} q, P_2 \rightarrow_{C} q}{P_1 \lor P_2 \rightarrow_{C} q} \)

A definition in Higher Order Logic

\[
P \rightarrow_{C} q = \hspace{1cm} (\exists r : P \rightarrow_{C} r \land r \rightarrow_{C} q) \lor \hspace{1cm} (\exists p_1, p_2 : (p = p_1 \lor p_2) \land p_1 \rightarrow_{C} q \land p_2 \rightarrow_{C} q)
\]
Proving the definition satisfies the axioms:

Prove that

1) \( p R t \ q \Rightarrow p R c \ q \)
2) \( p R c \ r \land r R c \ q \Rightarrow p R c \ q \)
3) \( p R c \ q \land p z R c \ q \Rightarrow p x \lor p z R c \ q \)

Trivially from the definition of \( R c \)

But more properties may be wanted in the logic.
A Primitive Recursive Definition of $R_{c}$

\[ \text{RelClose} \ 0 \ \text{R}_0 \ p \ q \ = \ p \ \text{R}_0 \ q . \ \top \]

\[ \text{RelClose} \ n+1 \ \text{R}_0 \ p \ q \ = \]

\[ (\exists r : \text{RelClose} \ n \ \text{R}_0 \ p \ r . \ \top \]

\[ \text{RelClose} \ n \ p \ r \ q . \ \top \]

\[ \text{RelClose} \ n \ p \ q \ r . \ ) \]

\[ \forall (\exists p_1 , p_2 : (p = p_1 \ \lor p_2) . \top \]

\[ \text{RelClose} \ n \ p_1 \ q . \ \top \]

\[ \text{RelClose} \ n \ p_2 \ q . \ ) \]

\[ p \ \text{R}_c \ q . \ = \ \exists n : \text{RelClose} \ n \ \text{R}_0 \ p \ q . \]

Again, prove that the three axioms and wanted theorems actually are satisfied.
Introducing bounds on progress

\[ \forall m, n : p \land (n = m) \rightarrow (p \land n \leq m) \lor q \]

\[ p \rightarrow q \]

We now need to generalise disjunction, since

the proof of the above theorems assumed

\[ (\exists m : M = m) \]

Hence, the disjunction theorem (axiom) becomes:

\[ \forall i : (\Pi i) \rightarrow q \]

\[ \rightarrow (\exists i : \Pi i) \rightarrow q \]
Generalising the definition of $R_C$

$$p \, R_C \, q \iff p \, R_T \, q \lor \left( \exists r : p \, R_C \, r \land r \, R_C \, q \right) \lor \left( \exists p : (p = \exists i : P_i) \land \left( \forall i : (P_i) \, R_C \, q \right) \right)$$

**NB**

*can we find a primitive recursive definition of this function:

$$\text{IntRelClose} \, \circ \, R_T \, p \, q = p \, R_T \, q \lor 1$$

$$\text{IntRelClose} \, n \, \text{IntRelClose} \, R_T \, p \, q =$$

$$\left( \exists p : (p = \exists i : P_i) \land \left( \forall i : \text{IntRelClose} \, n \, R_T \, (P_i) \, q \right) \right)$$

$$p \, R_C \, q = \exists n \, \text{IntRelClose} \, n \, R_T \, p \, q$$

**NB:** since we can't prove

$$\left( \forall i : (P_i) \, R_C \, q \right) \Rightarrow \left( \exists i : P_i \right) \, R_C \, q$$
A pure relational approach to $R_c$

$GenRelClose \ R =$

$(\forall p,q : \ p \mathbin R q \Rightarrow p \mathbin R q) \land$

$(\forall p,q : \ (\exists r : \ p \mathbin R r \land r \mathbin R q) \Rightarrow p \mathbin R q) \land$

$(\forall p,q : \ (\exists \iota : \ (p = \exists i : P_i) \land$

$(\forall \iota : \ (P_i) \mathbin R q) \Rightarrow p \mathbin R q) \land$

\[ p \mathbin R_c q = \forall r: GenRelClose \ R \Rightarrow p \mathbin R q \]

This definition actually satisfies the base axioms, but unfortunately we cannot prove

$\begin{array}{c}
p \mathbin R_c \text{false} \\
\hline
\end{array}$

However we do have:

$GenRelClose \ R_c$

$\begin{array}{c}
p \mathbin R_c q = p \mathbin R q \lor (\exists r : p \mathbin R_c r \land r \mathbin R q) \\
\lor (\exists \iota : (P_i) \mathbin R q) \\
\lor (\forall \iota : (P_i) \mathbin R_c q) \\
\end{array}$
How UNITY is derived.

1) Take unless, ensures as defined before.

2) Specialise the function which models the transitive and disjunctive closure of "single" progress relations (RT) by the ensures relation.

3) Prove all the missing theorems valid for UNITY.

4) Make (define) the appropriate representation of UNITY - programs as state transitions.

5) thats it!
Conclusion

- A complete UNITY system, except for the shown induction theorem, and with only finite disjunction has been implemented using the primitive recursive function \textit{RelClose} for modelling \textit{leadsTo}.

- It is an open question (to me) whether infinite disjunction and hence the progress induction theorem is actually sound.

- It was discovered that the substitution rule is only valid if the logic primitives \underline{unless}, \underline{ensures} and \underline{leadsTo} are restricted to the states reachable by the program. This means that HOL-UNITY doesn't have this rule!
Solving Reflexive Domain Equations in HOL

Kim Dam Petersen

TFL

kimdom @ tfl.dk
Contents

- Reflexive Domain Equations
- Methods for solving reflexive domain equations
- $D_\infty$
- Type free $\lambda$ calculus
- $P^\omega$
- What has been done
- What needs to be done
Reflexive Domain Equations

- Reflexive domain equations has the form: $D = \mathcal{E}(D)$, where $\mathcal{E}$ is composed from basic domains Bottom, Boolean and Number by Point, Pair, Union and Function contructions.

- A solution of a domain equation is a domain $D$, which is isomorphic to $\mathcal{E}(D)$.

- Example: $C = * \rightarrow C \rightarrow **$
Method for solving reflexive domain equations

- Use the $D_\infty$ domain constructions [Schmidt]
- Construct type free $\lambda$ calculus [Barendregt]
- Construct $P^\omega$ [Barendregt]
\[ D_\infty \text{ construction [Schmidt]} \]

- A domain is a (pointed) cpo
- All domains are contained in a universe \( D \) of domains
- A valid Domain Scheme \( \mathcal{E} : D \rightarrow D \) maps pointed cpos to pointed cpos
- Construct a sequence of domains \( D_n \) by:
  \[ D_0 = \{ \perp \} \text{ and } D_{n+1} = \mathcal{E}(D_n), \ n \geq 0 \]
- A limit \( D_\infty \) of the domain sequence exists
- A pair of continuous functions \( (\Phi, \Psi) : D_\infty \leftrightarrow \mathcal{E}(D_\infty) \) exists, such that: \( \Psi \circ \Phi = \text{Id}_{D_\infty} \) and \( \Phi \circ \Psi = \text{Id}_{\mathcal{E}(D_\infty)} \)
- Hence \( D_\infty \) is isomorphic to \( \mathcal{E}(D_\infty) \)
HOL implementation problems

- Constructing an object $\textit{Domain}$ that represents the universe of all domains

- $\textit{Domain}$ should contain the basic domains Bottom, Number and Boolean

- $\textit{Domain}$ should be closed wrt. domain constructions Point, Pair, Union and (continuous) Function
Type free $\lambda$ calculus

- Numbers, Booleans and Bottom are representable as closed $\lambda$ expressions.
- Pairs, Unions and Functions may be represented as closed $\lambda$ expression, if their components are
- Hence Domain may be represented as subsets of closed (type free) $\lambda$ expressions
Constructing type free $\lambda$ calculus [Barendregt]

- A reflexive cpo may be used to construct a model of type free $\lambda$ calculus
- A cpo $(D, \sqsubseteq)$ is reflexive if there exists two functions $f : D \to [D \to D]$ and $g : [D \to D] \to D$ such that: $f \circ g = \operatorname{id}_{[D \to D]}$
The $P^\omega$ model of $\lambda$ calculus [Barendregt]

- $P^\omega = (\{x|x \subseteq \mathbb{N}_0\}, \subseteq)$
- $[n, m] \leftrightarrow \frac{1}{2}(n + m)(n + m + 1) + m$
- $\{k_0, \ldots, k_{m-1}\} \leftrightarrow \Sigma_{i<m} 2^{k_i}$
- $\text{graph}(f) = \{[n, m]|m \in f(e_n)\}$
- $\text{graph} : [P^\omega \rightarrow P^\omega] \rightarrow P^\omega$ is continuous
- $\text{fun}(u)(x) = \{m|\exists e_n \subseteq x, [n, m] \in u\}$
- $\text{fun} : P^\omega \rightarrow [P^\omega \rightarrow P^\omega]$ is continuous
- $\text{fun}(\text{graph}(f)) = f$ for any continuous function $f$
- Hence $P^\omega$ is a reflexive cpo
- Hence $P^\omega$ may be used to construct Domain
- Hence $P^\omega$ may be used to solve reflexive domain equations
What needs to be done

- Completion of the $\lambda$ model
- Mapping basic domain values (Bottom, Booleans and Numbers) into $\lambda$ expressions
- Mapping constructed domain values (Pairs, Unions and Functions) into $\lambda$ expressions
- Solve reflexive domain equations using $\lambda$ expressions
- Make a type package for reflexive domains
What has been done in HOL

- Theories for: cpo, topology etc. has been defined
- Properties concerning $P^\omega$, fun and graph has been proved
- A $\lambda$ model based on $P^\omega$ is being constructed
Towards a HOL theory for
Finite Wordlength Arithmetic ...

Wim Ploegaerts†

Imec v.z.w.
Kapeldreef 75
3000 Leuven
- Belgium -

†Research Assistant with the Belgian National Fund for Scientific Research
Work partly sponsored by the CHEOPS-BRA
0. Introduction

1. Linking HOL to CATHEDRAL ...
2. The Missing Link ...
3. Some list theory ...
4. Integers: "zet" and more ...
5. The link ...
6. Conclusions ...

Linking hardware behavior to its integer equivalent

useful extensions of the library "zet"

which are available and documented
1. Linking HOL to CATHEDRAL...

**CATHEDRAL-II**


---

```plaintext
func main(var in WORD): out: WORD =
begin
out1 = WORD(coef3 + biquad(in, coef1, coef2, coef3, coef4));
in1 = out1;
out2 = WORD(coef5 + biquad(in2, coef10, coef20, coef30, coef40));
in2 = out2;
out = WORD(coef500 + biquad(in3, coef100, coef200, coef300, coef400));
end;

func biquad(in: WORD, b0, b1, c0, c1: num): WORD =
begin
state = in - WORD(state81'c1) + WORD(state82'c0);
return = state + WORD(state81'c0) + WORD(state82'c0);
end;
```
1. Linking HOL to CATHEDRAL...

CATHEDRAL...

Synthesis Environment for the design of ASIC's for DSP applications

- DSP = audio (CD), video, ...
- Optimizing results by limiting the application domain
- Cathedral 1,2,3,...

"From an idea to layout in 1 day..."

HOL...

The CHEOPS Project:

Setting up a Verification Environment for the CATHEDRAL results

Therefore

- Realistic Complexity ...
- Automatic versus Soundness...
  - A "designer" does not want to be confronted with HOL
  - General methodology instead of single proof
2. The Missing Link ...

High Level Silage
out = a + b

Bit True Silage
out: m = a:n + b:n

IMP 1

IMP 2

30 + 27 = 57
OVERFLOW

30 + 27 = 7

011110
+ 011011
111001
[b1; b2; ...; bn]
[a1; a2; ...; bn]
[o1; o2; ...; cm]

imec vzw
HOL Users Meeting
WP-2-10-90
2. The Missing Link ...

Proving integer expressions equal

Data Abstraction + Case split

IMP1

IMPi = {Abstract Operations}

IMP2

Prerequisites...
- powerful library on integer numbers,
  -> importance of "2"
- a tactic to prove that integer expressions are equal
- list-theory (variable length)
- the link from bitstrings to integers

What is available in HOL...
- list ...
- "zet" (T. Kalker)

What is needed...
2. The Missing Link ...

Implementation of
"Representational and Denotational Semantics of Digital Systems" (R.T. Boute)

- theory of number representations
- "detailed" transformational reasoning style
- functional formalism closely related to HOL

```
4
 MSB 1 0 0
 LSB 0 0 1
```

du : (bool)string -> (num)

\[(du [] = 0) \land (du (CONS a x)) = a' + 2 \times (du x)\]

ru : (num) -> (bool) string

\[(ru 0 = []) \land (ru x = (CONS (x \text{ mod } 2) (ru (x \text{ div } 2))))\]

ds : (bool)string -> (set)

\[(ds [] = 0) \land (ds (APPEND x [s]) = (du x) - s' \times 2)\]

rs : (set) -> (bool) string

\[(rs 0 = [F]) \land (rs -1 = [T]) \land (rs m = (CONS (m \text{ mod } 2) (rs (m \text{ div } 2))))\]
3. Some List theory...

<table>
<thead>
<tr>
<th>a1; a2; ...; a(n-1); a; an</th>
</tr>
</thead>
<tbody>
<tr>
<td>HD</td>
</tr>
<tr>
<td>TL</td>
</tr>
<tr>
<td>TL</td>
</tr>
<tr>
<td>HL</td>
</tr>
<tr>
<td>LAST</td>
</tr>
</tbody>
</table>

\[ \begin{align*}
\text{H}\text{D} & = x \\
(fa1) & = f(a1) \\
(\text{h}: f, a, (\text{APPEND } [h]) & = f(f(a), h))
\end{align*} \]

- Definitions
- induction tactic
- relation with constructs of "list"

Use: du, ds

[!] y : \text{LAST} (\text{APPEND } [y]) = y

[!] y : \text{HL} (\text{APPEND } [y]) = 1
4. Integers: "zet" and more ...

"zet" (T.Kalker)

- formalization of integers
  - zero (0), een (1)
- addition, subtraction, multiplication, less, leq
- SIGMA (Σ), PI (π)
- maximum and minimum of a subset
- recursive definitions in zet
- induction on integers
- several useful theorems, but ...

extensions of "zet"

- absolute value and sign of an integer
- positive power of an integer
- integer division and remainder

|- ! m n.

~ (n = 0) =>

(m = ((m div n) * n) + (m mod n)) ∧
(0 ≤ (m mod n)) ∧ ((m mod n) < |n|)

-4 div 3 = -2  -4 mod 3 = 2
4. Integers: "zet" and more ...

- concept of even and odd integers

- div2, mod2, pow2

\[ \!n. (0 < n) \implies (?! k. (2 \leq n) \land (n < 2^k)) \]

- pos_log : that "k" for which ....

- log2 : related to div2

\[ \!n. (\text{log2} 0 = 0) \land (\text{log2} -1 = 0) \land \\
(\neg(n = 0) \land \neg(n = -1) \implies \\
(\text{log2} n = \text{SUC} (\text{log2} (\text{div2} n)))) \]

INDUCTION TACTIC

\[ !Q. ((Q 0) \land (Q -1)) \land \\
(\!m. Q m \implies Q (2m)) \land \\
(\!m. Q m \implies Q (2m + 1)) \]
\[ \implies (!n. Q n) \]

Related to bitstrings:

100 \leftrightarrow 4

1000 \leftrightarrow 2*4 1001 \leftrightarrow 2*4 + 1
4. Integers: "zet" and more ...

\[(a + b)(d + e) - db = eb + ad + ae\]

"m + n = k + l"

["m - k = l - n"]

"a < b + c"

["a - b ≤ c - 1"]

**NORMAL_FORM_CONV**

Make normalized sum of products

\[\exp = p1 + (p2 + (p3 + \ldots))\]

\[\pi = a1 * (a2 * (a3 * \ldots))\]

\[= - (\ldots)\]

\[ai = \text{not an addition, subtraction or multiplication}\]

**make syntactical normal form**

use of "<<" and bubble sorting to order

- subterms of \(\pi\) (ai)
- ordered subterms of \(\exp\)
5. The Link ...

\[ \text{du} : (:\text{bool})\text{string} \to (:\text{set}) \]
\[ (\text{du} [] = 0) \land (\forall x . \text{du} (\text{CONS} \ a \ x)) = a' + 2 \times (\text{du} \ x) \]

\[ \text{ru} : (:\text{set}) \to (:\text{bool}) \text{string} \]
\[ \forall n . (0 \leq n) \implies \]
\[ (\text{ru} \ 0 = []) \land (\text{ru} \ n = (\text{CONS} \ (n \ \text{mod} \ 2) \ (\text{ru} \ (n \ \text{div} \ 2)))) \]

\[ \text{ds} : (:\text{bool})\text{string} \to (:\text{set}) \]
\[ (\text{ds} [] = 0) \land (\forall x . \text{ds} (\text{APPEND} \ x \ [s])) = (\text{du} \ x) - s' \ 2 \]

\[ \text{rs} : (:\text{set}) \to (:\text{bool})\text{string} \]
\[ (\text{rs} \ 0 = [\text{F}]) \land (\text{rs} \ -1 = [\text{T}]) \land \]
\[ (\forall m . (m = 0) \land (m = -1)) \implies \]
\[ (\text{rs} \ m = (\text{CONS} \ (m \ \text{mod} \ 2) \ (\text{rs} \ (m \ \text{div} \ 2)))) \]
5. Conclusions ...

The following theories are available and documented:

- general purpose extensions of the library "zet": div, mod, pow (documentation of "zet" and some tactics and conversions of "auxiliary" included)
- a normalization conversion to prove integer expressions equal
- extensions of "zet" for the special case "2": div2, mod2, pow2, log2

The theory on the denotation and representation functions is available but not yet documented.

These theories are needed for the characterization of finite wordlength arithmetic.
More Reasons
Why Higher-Order Logic is a Good Formalism
for Specifying and Verifying Hardware

Jeff Joyce
University of British Columbia

October 1990
"Why higher-order logic?"
Mike Gordon (September 1985) . . .

"Why Higher-Order Logic is a
Good Formalism for Specifying and Verifying Hardware"

(Cambridge Tech. Report No. 77)

. . . the opening paragraph reads:

The purpose of this paper is to show, via examples, that:

1. Many kinds of digital systems can be formally specified using the notation
   of formal logic; specialized hardware descriptions are not needed.

2. The inference rules of logic provide a practical means of proving systems
   correct; specialized deductive systems are not needed.
Two Reasons "Why Higher-Order Logic?"

In this talk, I propose two reasons why higher-order logic is a good formalism for specifying and verifying hardware:

1. generic specification
2. embedded formalisms
Outline of this Talk ... 

1. Uses of Higher-Order Functions
   - some common uses
   - some more sophisticated uses

2. Alternatives to h.o.l.

3. Arguments against h.o.l.

4. Generic Specification
   - simple example
   - microprocessor specification

5. Embedding Other Formalisms
   - temporal logic

6. Conclusions
Some Common Uses of Higher-Order Functions ...

... when specifying hardware with predicates,

1. signals as functions of time

\[
\text{AndGate}(a', b', out)
\]

\[
a, b, out : \text{time} \rightarrow \text{bool}
\]
2. signals as functions of position

\[
\text{RippleCarry } n \ (a, b, c_{in}, s_{umout})
\]

\[
a, b, \text{sum} : \text{position} \rightarrow \text{bool}
\]

\[
c_{in}, c_{out} : \text{bool}
\]
More Sophisticated Uses of Higher-Order Functions...

abstract time: 

f: 

concrete time: 

g: F F T F F F T F T F 

...defining parameterized timing relationships,

\[
\text{TimeOf (g:time->bool)} 
\]

Define ( 
"(TimeOf g o = e.t. First g t) \land 
(TimeOf g (SUC u) = e.t. Next g (TimeOf g u, t)))"));

Define ("First g t = (\forall p. p < t \implies \neg (g p)) \land (g t)"));

Define ( 
"Next g (t1, t2) = t1 < t2 \land (\forall t. t1 < t \land t < t2 \implies \neg (g t)) \land (g t2)"));

...and deriving generalized theorems,
For example, ... 

the generalized theorem,

\[ \forall g. x. \]
\[ (\exists t. g t) \land (\forall t. g t \implies \exists m. \text{Next } g(t, t+m) \land r(t, t+m)) \implies\]
\[ \forall u. \text{TimeOf } g u, \text{TimeOf } g(u+1) \]

can be used to reduce the problem of proving,

\[ \forall u. r(\text{TimeOf } g u, \text{TimeOf } g(u+1)) \]

to a pair of simpler problems:

\[ \exists t. g t \]
\[ \forall t. g t \implies \exists m. \text{Next } g(t, t+m) \land r(t, t+m) \]

This generalized result provides the basis for a HOL tactic:

\[ \forall u. r[\text{TimeOf } g u, \text{TimeOf } g(u+1)] \]

\[ \exists t. g t, \quad \forall t. g t \implies \exists m. \text{Next } g(t, t+m) \land r[t, t+m] \]
Alternatives to Higher-Order Logic, ...

... as a hardware specification language:

- Conventional HDL's
  - generally oriented to simulation
  - limited ability to express behaviour
  - may lack underlying formal semantics
- Special Purpose Formalisms
  - fixed ideas about hardware
- First-Order Logic
  - sometimes expressibility is compromised further other goals
  - e.g. disallowed "∃" to make specifications executable
  - may have extra constructs reminiscent of h.o.l.
    - e.g. parameterized specifications in OBJ
- Type Theory
  - dependent types, but undecidable
Arguments Against the Higher-Order Approach ...

- notation is difficult
- undecidability concerns
- proofs are much harder
- expressibility not needed
- specifications not executable
- the "straw man" attack
To give you an idea of their flavour ...

... with respect to higher-order programming, one author writes:

"Unfortunately, the logic of higher-order functions is difficult, and in particular, higher order unification is undecidable. Moreover (and closely related), higher order expressions are notoriously difficult for humans to read and write correctly."

... and then, the same author goes on to say:

"Although higher-order logic cannot always be avoided in specification and verification, it should be avoided whenever possible, for the same reasons as in programming."

Quotations from:

Generic Specification

In 1989, under contract to NASA, CLI sub-contracted the task of writing a review of the HOL system to David Musser of Rensselaer Polytechnic.

In his report, Musser wrote:

"The major weakness of HOL appears to be the lack of effective support for constructing specifications and proofs at a high level of abstraction, ..."

and more specifically,

"HOL is also lacking in packaging features that would help in structuring large specifications, such as those of VIPER or more complex microprocessors"
Musser compared HOL to SRI's EHDM system which,

"...provides a parameterized module capability that permits structuring specifications as modules that can be instantiated in many different ways (similarly to Ada generic packages and subprograms)."

In this part of my talk, I will argue that:

1. generic specification does indeed offer many benefits
2. HOL logic can directly express genericity
   ...without additional constructs!
Advantages of Generic Specification ...

In addition to modularity, abstraction and re-usability, generic description can be used to filter out non-essential detail in the context of formal proof.

Filtering out non-essential details, ...

- sharpens the distinction between what has and what has not been formally considered in a correctness proof.
- supports a truly hierarchical approach to the formal verification of digital circuits where each level in a hierarchical specification is isolated from details only relevant to other levels.
- reduces the amount of special-purpose infrastructure needed to reason about particular application areas, e.g., hardware-oriented data types.
For example, ...

The formal specification of the 32-bit Viper major-state machine made extensive use of special-purpose data types and constants:

   e.g., :word4, :word32, VAL4, WORD32

Consequently, these correctness results:

- were not as general as possible

   e.g. what about a 64-bit version?

- needed clarification about what aspects of the computational model were formally considered:

  "There was no computation of values at the major state level - that is, additions, comparisons, shifts, and so on - so the essential correctness of Viper was not really addressed; the proof did not require any analysis of the function representing the arithmetic-logic unit, at either level".

- could not be easily re-produced in other verification systems lacking special support for reasoning about hardware
External View of the Resettable Counter,

![External View Diagram]

Internal View of the Resettable Counter,

![Internal View Diagram]
A First Attempt, ...

\[
\begin{align*}
\text{def } \text{MUX} & \text{ (reset, i, out)} = \forall t. \text{ out } t = (\text{reset } t \Rightarrow 0 \mid (i \ t)) \\
\text{def } \text{REG} & \text{ (i, out)} = \forall t. \text{ out } (t+1) = i \ t \\
\text{def } \text{INC.} & \text{ (i, out)} = \forall t. \text{ out } t = ((i \ t) + 1) \\
\text{def } \text{COUNT_IMP} & \text{ (reset, out)} = \\
& \exists p1 \ p2. \\
& \text{MUX (reset, p1, p2)} \land \\
& \text{REG (p2, out)} \land \\
& \text{INC (out, p1)} \\
\text{def } \text{COUNT} & \text{ (reset, out)} = \\
& \forall t. \text{ out } (t+1) = (\text{reset } t \Rightarrow 0 \mid ((\text{out } t) + 1)) \\
\text{thm } \text{COUNT_IMP} & \text{ (reset, out)} \Rightarrow \text{COUNT} \text{ (reset, out)} \\
\end{align*}
\]

The More Detailed, The Better？

\[
\begin{align*}
\text{def } \text{MUX} & \text{ (n, reset, i, out)} = \forall t. \text{ out } t = (\text{reset } t \Rightarrow 0 \mid (i \ t)) \\
\text{def } \text{REG} & \text{ (n, i, out)} = \forall t. \text{ out } (t+1) = i \ t \\
\text{def } \text{INC} & \text{ (n, i, out)} = \forall t. \text{ out } t = (((i \ t) + 1) \mod 2^n) \\
\text{def } \text{COUNT_IMP} & \text{ (n, reset, out)} = \\
& \exists p1 \ p2. \\
& \text{MUX (n, reset, p1, p2)} \land \\
& \text{REG (n, p2, out)} \land \\
& \text{INC (n, out, p1)} \\
\text{def } \text{COUNT} & \text{ (n, reset, out)} = \\
& \forall t. \text{ out } (t+1) = (\text{reset } t \Rightarrow 0 \mid (((\text{out } t) + 1) \mod 2^n)) \\
\text{thm } \text{COUNT_IMP} & \text{ (n, reset, out)} \Rightarrow \text{COUNT} \text{ (n, reset, out)} \\
\end{align*}
\]
Parameterizing with Function Variables, ...

\[ \text{\texttt{\texttt{\texttt{\texttt{$\vdash_{def}$ MUX (inc) (reset,i,out) = }}}} \forall_t \text{ out } t = (\text{reset } t \Rightarrow 0 | (i \ t))} \]

\[ \text{\texttt{\texttt{\texttt{\texttt{$\vdash_{def}$ REG (inc) (i,out) = }}}} \forall_t \text{ out } (t+1) = i \ t} \]

\[ \text{\texttt{\texttt{\texttt{\texttt{$\vdash_{def}$ INC (inc) (i,out) = }}}} \forall_t \text{ out } t = \text{inc } (i \ t)} \]

\[ \text{\texttt{\texttt{\texttt{\texttt{$\vdash_{def}$ COUNT_IMP (inc) (reset,out) = }}}} \exists_p \ p_2.} \]

\[ \text{MUX } (\text{inc}) \ (\text{reset,p1,p2}) \land \]

\[ \text{REG } (\text{inc}) \ (\text{p2,out}) \land \]

\[ \text{INC } (\text{inc}) \ (\text{out,p1}) \]

\[ \text{\texttt{\texttt{\texttt{\texttt{$\vdash_{def}$ COUNT (inc) (reset,out) = }}}} \forall_t \text{ out } (t+1) = (\text{reset } t \Rightarrow 0 | (\text{inc } (\text{out } t)))} \]

\[ \text{\texttt{\texttt{\texttt{\texttt{$\vdash_{thm}$ COUNT_IMP (inc) (reset,out) \Rightarrow COUNT (inc) (reset,out)}}}}} \]

Parameterizing with Type Variables, ...

\[ \text{\texttt{\texttt{\texttt{\texttt{$\vdash_{def}$ MUX (inc,zero) (reset,i,out) = }}}} \forall_t \text{ out } t = (\text{reset } t \Rightarrow \text{zero } | (i \ t))} \]

\[ \text{\texttt{\texttt{\texttt{\texttt{$\vdash_{def}$ REG (inc,zero) (i,out) = }}}} \forall_t \text{ out } (t+1) = i \ t} \]

\[ \text{\texttt{\texttt{\texttt{\texttt{$\vdash_{def}$ INC (inc,zero) (i,out) = }}}} \forall_t \text{ out } t = \text{inc } (i \ t)} \]

\[ \text{\texttt{\texttt{\texttt{\texttt{$\vdash_{def}$ COUNT_IMP (inc,zero) (reset,out) = }}}} \exists_p \ p_2.} \]

\[ \text{MUX } (\text{inc,zero}) \ (\text{reset,p1,p2}) \land \]

\[ \text{REG } (\text{inc,zero}) \ (\text{p2,out}) \land \]

\[ \text{INC } (\text{inc,zero}) \ (\text{out,p1}) \]

\[ \text{\texttt{\texttt{\texttt{\texttt{$\vdash_{def}$ COUNT (inc,zero) (reset,out) = }}}} \forall_t \text{ out } (t+1) = (\text{reset } t \Rightarrow \text{zero } | (\text{inc } (\text{out } t)))} \]

\[ \text{\texttt{\texttt{\texttt{\texttt{$\vdash_{thm}$ COUNT_IMP (inc,zero) (reset,out) \Rightarrow COUNT (inc,zero) (reset,out)}}}}} \]
Using "Representation Variables", ...

\[ \begin{align*}
\text{def} & \quad \text{inc rep} = \text{FST rep} \\
\text{def} & \quad \text{zero rep} = \text{SND rep} \\
\text{def} & \quad \text{MUX (rep) (reset,i,out)} = \forall t. \text{out } t = (\text{reset } t \Rightarrow (\text{zero rep}) | (i t)) \\
\text{def} & \quad \text{REG (rep) (i,out)} = \forall t. \text{out } (t+1) = i t \\
\text{def} & \quad \text{INC (rep) (i,out)} = \forall t. \text{out } t = (\text{inc rep}) (i t) \\
\text{def} & \quad \text{COUNT_IMP (rep) (reset,out)} = \\
& \quad \exists p1 p2. \\
& \quad \text{MUX (rep) (reset,p1,p2)} \land \\
& \quad \text{REG (rep) (p2,out)} \land \\
& \quad \text{INC (rep) (out,p1)} \\
\text{def} & \quad \text{COUNT (rep) (reset,out)} = \\
& \quad \forall t. \text{out } (t+1) = (\text{reset } t \Rightarrow (\text{zero rep}) | ((\text{inc rep}) (out t))) \\
\end{align*} \]

\[ \begin{align*}
\text{rep: } ((\text{word} \rightarrow \text{word}) \times \text{word}) \\
\text{inc} & \quad \forall \text{lo} \\
\text{inc rep) - "the increment operation"} \\
\text{zero rep) - "the representation of zero"} \\
\h_{\text{thm}} & \quad \text{COUNT_IMP (rep) (reset,out)} \Rightarrow \text{COUNT (rep) (reset,out)}
\end{align*} \]
This correctness theorem is fully generic:

$$\vdash_{thm} COUNT_{-}IMP \ (rep) \ (reset, out) \implies COUNT \ (rep) \ (reset, out)$$

Recall Musser’s remark that SRI’s EHDM system,

“...provides a parameterized module capability that permits structuring specifications as modules that can be instantiated in many different ways.”

This is quite straightforward with representation variables,

...here are two simple examples,
Example 1: Idealized Counter

Here again is the idealized counter:

$$\text{REP\_num} = (\lambda x. x + 1, 0)$$

$$\vdash \text{thm \ COUNT\_IMP \ (REP\_num) \ (reset, out)} \Rightarrow \text{COUNT \ (REP\_num) \ (reset, out)}$$

...which can be expanded to:

$$\vdash \text{thm \ COUNT\_IMP \ (REP\_num) \ (reset, out)} \Rightarrow \forall t. \ text{out} (t+1) = (\text{reset} \ t \Rightarrow 0 \mid ((\text{out} \ t) + 1))$$

Example 2: 8-bit Counter

Another instance could be an 8-bit version based on built-in HOL types:

$$\text{REP8} = (\lambda x. \text{WORD8} (((\text{VAL8} \ x) + 1), \text{WORD8} \ 0)$$

$$\vdash \text{thm \ COUNT\_IMP \ (REP8) \ (reset, out)} \Rightarrow \text{COUNT \ (REP8) \ (reset, out)}$$

...which can be expanded to:

$$\vdash \text{thm \ COUNT\_IMP \ (REP8) \ (reset, out)} \Rightarrow \forall t. \ text{out} (t+1) = (\text{reset} \ t \Rightarrow \text{WORD8} \ 0 \mid (\text{WORD8} (((\text{VAL8} \ x) + 1))))$$
Generic Specification of a Simple Microprocessor ...

**TAMARACK-3**

- **datain** - data from memory
- **dack** - data acknowledge
- **idle** - extended cycle mode
- **ireq** - interrupt request
- **wmem** - read/write select
- **dataout** - data to memory
- **dreq** - data request
- **addr** - address to memory
- **iack** - interrupt acknowledge
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode Value</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>JZR</td>
<td>0</td>
<td>jump if zero</td>
</tr>
<tr>
<td>JMP</td>
<td>1</td>
<td>jump</td>
</tr>
<tr>
<td>ADD</td>
<td>2</td>
<td>add accumulator</td>
</tr>
<tr>
<td>SUB</td>
<td>3</td>
<td>subtract accumulator</td>
</tr>
<tr>
<td>LDA</td>
<td>4</td>
<td>load accumulator</td>
</tr>
<tr>
<td>STA</td>
<td>5</td>
<td>store accumulator</td>
</tr>
<tr>
<td>RFI</td>
<td>6</td>
<td>return from interrupt</td>
</tr>
<tr>
<td>NOP</td>
<td>7</td>
<td>no operation</td>
</tr>
</tbody>
</table>

JZR - jump if zero
\[ pc \leftarrow \text{if iszero acc then inst else inc pc} \]

JMP - jump
\[ pc \leftarrow \text{inst} \]

ADD - add accumulator
\[ \text{acc} \leftarrow \text{add (acc, operand)} \]
\[ pc \leftarrow \text{inc pc} \]

SUB - subtract accumulator
\[ \text{acc} \leftarrow \text{sub (acc, operand)} \]
\[ pc \leftarrow \text{inc pc} \]

LDA - load accumulator
\[ \text{acc} \leftarrow \text{operand} \]
\[ pc \leftarrow \text{inc pc} \]

STA - store accumulator
\[ \text{mem} \leftarrow \text{store (mem, address inst, acc)} \]
\[ pc \leftarrow \text{inc pc} \]

RFI - return from interrupt
\[ pc \leftarrow \text{rtn} \]
\[ \text{iack} \leftarrow \text{F} \]

NOP - no operation
\[ pc \leftarrow \text{inc pc} \]

Interrupt
\[ \text{if iack = F then} \]
\[ pc \leftarrow 0 \]
\[ \text{rtn} \leftarrow \text{pc} \]
\[ \text{iack} \leftarrow \text{T} \]

212
Data Types:

:bool - Boolean values \{T,F\}
:num - natural numbers \{0,1,2,\ldots\}
:*wordn - full-size machine words
:*word3 - instruction opcodes
:*word4 - 4-bit words
:*address - memory addresses
:*memory - memory states

Generic Functions:

(iszero rep) - “test if zero”
(inc rep) - “increment”
(add rep) - “addition”
(sub rep) - “subtraction”
(wordn rep) - “full-size word representation of a number”
(valn rep) - “value of a full-size word”
(opcode rep) - “extract opcode field”
(val3 rep) - “value of an opcode”
(address rep) - “extract address field”
(fetch rep) - “read memory”
(store rep) - “write memory”
(word4 rep) - “value of a 4-bit word”
(val4 rep) - “4-bit word representation of a number”
rep_ty =
  (*wordn→bool)× % iszero %
  (*wordn→*wordn)× % inc %
  (*wordn×*wordn→*wordn)× % add %
  (*wordn×*wordn→*wordn)× % sub %
  (num→*wordn) × % wordn %
  (*wordn→num) × % valn %
  (*wordn→word3)× % opcode %
  (*word3→num)× % val3 %
  (*wordn→*address)× % address %
  (*memory×*address→*wordn)× % fetch %
  (*memory×*address×*wordn→*memory)× % store %
  (num→*word4)× % word4 %
  (*word4→num) % val4 %
\[ \text{mtime}(\text{rep}, \text{rep_ty}) = \]
\[ \text{TamarackImpr}(\text{rep}, \text{rep_ty}) (\text{datain}, \text{dack}, \text{idle}, \text{ireq}, \text{mpc}, \text{mar}, \text{pc}, \]
\[ \text{acc}, \text{ir}, \text{rtn}, \text{arg}, \text{buf}, \text{iack}, \text{dataout}, \text{wmem}, \text{dreq}, \text{addr}) = \]
\[ \exists \text{zroflag opc cntls.} \]
\[ \text{Cnt1Unit rep (dack, idle, ireq, iack, zroflag, opc, mpc, cntls}) \land \]
\[ \text{DataPath rep (cntls, datain, mar, pc, acc, ir, rtn, iack, arg, buf, dataout, wmem, dreq, addr, zroflag, opc))} \]
\[ \text{bus busokay alu pwr gnd rmem wmar wpc rpc wacc racc wir ir rwrn rrrn warg alu0 alui rbuf.} \]
\[ \text{DecodeCntls (cntls, wmem, rmem, wmar, rpc, rpc, wacc, racc, wir, ir, rwrn, rrrn, warg, alu0, alui, rbuf))} \land \]
\[ \text{BusOkay (rmem, rpc, racc, ir, rrrn, rbuf, busokay}) \land \]
\[ \text{Interface rep (busokay, wmem, rmem, bus, datain, dataout))} \land \]
\[ \text{OR (wmem, rmem, dreq))} \land \]
\[ \text{Register (busokay, wmar, gnd, bus, bus, mar))} \land \]
\[ \text{AddrField rep (mar, addr))} \land \]
\[ \text{Register (busokay, wpc, rpc, bus, bus, pc))} \land \]
\[ \text{Register (busokay, wacc, racc, bus, bus, acc))} \land \]
\[ \text{TestZero rep (acc, zroflag))} \land \]
\[ \text{Register (busokay, wir, ir, bus, bus, ir))} \land \]
\[ \text{OpcField rep (ir, opc))} \land \]
\[ \text{Register (busokay, wrtn, rrrn, bus, bus, rtn))} \land \]
\[ \text{JKFF (rwrn, rrrn, iack))} \land \]
\[ \text{Register (busokay, warg, gnd, bus, bus, arg))} \land \]
\[ \text{ALU rep (alu0, alui, arg, bus, alu))} \land \]
\[ \text{Register (busokay, pwr, rbuf, alu, bus, buf))} \land \]
\[ \text{PWR pwr} \land \]
\[ \text{GND gnd} \]
\[ \text{\text{mtime}(\text{rep}, \text{rep_ty}) (f0, f1, inp1, inp2, out)) = } \]
\[ \text{out t = (((f0 t, f1 t) = (T, T)) \rightarrow (((\text{inc rep}) (\text{inp2 t})) \land \]
\[ (((f0 t, f1 t) = (T, F)) \rightarrow (((\text{add rep}) (\text{inp1 t, inp2 t})) \land \]
\[ (((f0 t, f1 t) = (F, T)) \rightarrow (((\text{sub rep}) (\text{inp1 t, inp2 t})) \land \]
\[ (((\text{wordn rep}) 0)))} \]
```plaintext
def TamarackBeh (rep, rep_ty) (ireq, mem, pc, acc, rtn, iack) =
  if time then
    (mem (u+1), pc (u+1), acc (u+1), rtn (u+1), iack (u+1)) =
    NextState rep (ireq u, mem u, pc u, acc u, rtn u, iack u)
  fi

def NextState (rep, rep_ty) (ireq, mem, pc, acc, rtn, iack) =
  let opcval = OpcVal rep (mem, pc) in
  ((ireq ∧ ¬iack) ⇒ IRQ_SEM rep (mem, pc, acc, rtn, iack) |
   (opcval = JZR_OPCODE) ⇒ JZR_SEM rep (mem, pc, acc, rtn, iack) |
   (opcval = JMP_OPCODE) ⇒ JMP_SEM rep (mem, pc, acc, rtn, iack) |
   (opcval = ADD_OPCODE) ⇒ ADD_SEM rep (mem, pc, acc, rtn, iack) |
   (opcval = SUB_OPCODE) ⇒ SUB_SEM rep (mem, pc, acc, rtn, iack) |
   (opcval = LDA_OPCODE) ⇒ LDA_SEM rep (mem, pc, acc, rtn, iack) |
   (opcval = STA_OPCODE) ⇒ STA_SEM rep (mem, pc, acc, rtn, iack) |
   (opcval = RFI_OPCODE) ⇒ RFI_SEM rep (mem, pc, acc, rtn, iack) |
   NOP_SEM rep (mem, pc, acc, rtn, iack))

def JZR_SEM (rep, rep_ty)
  (mem:*memory, pc:*wordn, acc:*wordn, rtn:*wordn, iack:bool) =
  let inst = (fetch rep) (mem, (address rep) pc) in
  let nextpc = ((iszero rep) acc ⇒ inst | ((inc rep) pc) in
    (mem, nextpc, acc, rtn, iack)

def JMP_SEM (rep, rep_ty)
  (mem:*memory, pc:*wordn, acc:*wordn, rtn:*wordn, iack:bool) =
  let inst = (fetch rep) (mem, (address rep) pc) in
    (mem, inst, acc, rtn, iack)

def ADD_SEM (rep, rep_ty)
  (mem:*memory, pc:*wordn, acc:*wordn, rtn:*wordn, iack:bool) =
  let inst = (fetch rep) (mem, (address rep) pc) in
    let operand = (fetch rep) (mem, (address rep) inst) in
    (mem, (inc rep) pc, (add rep) (acc, operand), rtn, iack)
```
- `∀datain pwr dataout wmem dreq addr.
  Val3_CASES_ASM (rep:rep_ty) ∧
  Val4Word4_ASM rep ∧
  TamarackImp rep (datain,pwr,pwr,ireq,mpc,mar,pc,
  acc,ir,rtn,arg,buf,iack,dataout,wmem,dreq,addr) ∧
  SynMemory rep (wmem,addr,dataout,mem,datain) ∧
  PWR pwr ∧
  (((val4 rep) o mpc) 0 = 0)

⇒
  let f = TimeOfCycle rep (ireq,mem,pc,acc,rtn,iack) in
  TamarackBeh rep (ireq o f,mem o f,pc o f,acc o f,rtn o f,iack o f)

- independent of any fixed word size.
- independent of elaborate proof infrastructure
- could be linked into a verified stack
Here is a representation for a 16-bit version,

```
Define ("ISZER016 w = ((VAL16 w) = 0)");
Define ("INC16 w = WORD16 ((VAL16 w) + 1)");
Define ("ADD16 (w1, w2) = WORD16 ((VAL16 w1) + (VAL16 w2))");
Define ("SUB16 (w1, w2) = WORD16 ((VAL16 w1) - (VAL16 w2))");
Define ("OPCODE w = WORD3 (V (SEG (0, 2) (BITS16 w)))");
Define ("ADDRESS w = WORD13 (V (SEG (3, 15) (BITS16 w)))");
Define ("REP16 =
    ISZER016, % iszero %
    INC16, % inc %
    ADD16, % add %
    SUB16, % sub %
    WORD16, % word %
    VAL16, % val %
    OPCODE, % opcode %
    VAL3, % val3 %
    ADDRESS, % address %
    (λ(x,y). FETCH13 x y), % fetch %
    (λ(x,y,z). STORE13 y z x), % store %
    WORD4, % word %
    VAL4")
```

...which is just "plugged into" the generic correctness theorem:

```
∀datain pwr dataout wmem dreq addr.
  Val3_CASES_ASM REP_16 ∧
  Val4Word4_ASM REP_16 ∧
  TamarackImp REP_16 (datain, pwr, pwr, ireq, mpc, mar, pc,
    acc, ir, rtn, arg, buf, iack, dataout, wmem, dreq, addr) ∧
  SynMemory REP_16 (wmem, addr, dataout, mem, datain) ∧
  PWR pwr ∧
  (((val4 rep) o mpc) 0 = 0)
  →
  let f = TimeOfCycle rep (ireq, mem, pc, acc, rtn, iack) in
  TamarackBeh REP_16 (ireq o f, mem o f, pc o f, acc o f, rtn o f, iack o f)
```
fully generic description of hardware in vhdl,

package TYPES is

    subtype word_3 is bit_vector (2 downto 0);
    subtype word_4 is bit_vector (3 downto 0);
    subtype word_n is bit_vector (15 downto 0);
    subtype num is natural;
    subtype bool is boolean;

end TYPES;

library tamarack;
use tamarack.types.all;

package TAM3 is

    function iszero (inp:word_n) return bool;
    function inc (inp:word_n) return word_n;
    function add (inp1,inp2:word_n) return word_n;
    function sub (inp1,inp2:word_n) return word_n;
    function wordn (inp:num) return word_n;
    function valn (inp:word_n) return num;
    function opcode (inp:word_n) return word_3;
    function val3 (inp:word_3) return num;
    function word4 (inp:num) return word_4;
    function val4 (inp:word_4) return num;

end TAM3;

function iszero (inp:word_n) return bool is
    variable result :bool := TRUE;
begin
    for i in inp'low to inp'high loop
        if inp(i) = '1' then
            result := FALSE;
        end if;
    end loop;
    return(result);
end iszero;
A typical formalism involves a set of operators,

\[
\begin{align*}
\Box & \quad \text{"henceforth"} \\
\Diamond & \quad \text{"eventually"} \\
\bigcirc & \quad \text{"next"} \\
\mathcal{U} & \quad \text{"until"}
\end{align*}
\]

... and a set of transformation rules:

\[
\begin{align*}
\text{e.g.,} \quad & \frac{P \land (\neg Q)}{P} \rightarrow (P \cup Q) \\
\text{e.g.,} \quad & \frac{P \quad (\not\!\not Q)}{\Box P}
\end{align*}
\]

To embed a formalism in HOL:

1. definitions are given for the operators based on a semantic theory for the formalism;

\[
\Box P = \lambda t. \forall n. P(t+n)
\]

2. standard transformation rules are derived from operator definitions as theorems of higher-order logic

\[
\forall P \ Q. \ \text{VALID}((P \land (\neg Q)) \rightarrow (\bigcirc P)) \rightarrow \text{VALID}(P \rightarrow (P \cup Q))
\]
Contrast with ‘Syntax-Based’ Approaches

This approach differs fundamentally from ‘syntax-based’ approaches of developing support tools for formalisms:

for instance, tools such as the Cornell Synthesizer Generator (CSG) have been used to develop support tools for the protocol specification language LOTOS.

In the ‘syntax-based’ approach, transformation rules are simply ‘programmed’ (perhaps inconsistently !) into the system.

…but in a HOL approach,

Transformation rules are be formally dervied as logical consequences of the semantic theory (i.e. definitions of primitive operators).

While the HOL system is principally concerned the semantic aspects of representing an embedding formalism, it is often possible to also represent syntactic aspects of the formalism.
An Example

Here's an example of when an embedded notations makes a big difference!

Synchronizing Data Transfer with Handshaking Signals
The timing diagram is the standard way to describe constraints on the relative order of events in a handshaking sequence.

Some of these constraints, along with additional constraints, are expressed below in natural language.

"whenever the request signal becomes true, it must remain true until it is acknowledged"

"every request must eventually be acknowledged"

"whenever the acknowledgement signal becomes true, it must remain true until the request signal returns to false"

"the request signal will eventually return to false after the request is acknowledged"

"whenever the request signal is false, it will remain false until the acknowledgement signal is also false"

"the acknowledgement signal will eventually return to false after the request signal returns to false"

"once false, the acknowledgement signal will remain false until there is a request"

"whenever the acknowledgement signal is false, there will eventually be a request"

For handshaking to work correctly, these constraints must hold.
The natural language descriptions are the previous page could be translated into the following set of eight assertions:

\[ \forall t. \text{req } t \implies (\forall n. (\forall m. m < n \implies \text{ack}(t + m)) \implies \text{req}(t + n)) \]

\[ \forall t. \text{req } t \implies (\exists n. \text{ack}(t + n)) \]

\[ \forall t. \text{ack } t \implies (\forall n. (\forall m. m < n \implies \text{req}(t + m)) \implies \text{ack}(t + n)) \]

\[ \forall t. \text{ack } t \implies (\exists n. \neg \text{req}(t + n)) \]

\[ \forall t. \text{req } t \implies (\forall n. (\forall m. m < n \implies \text{ack}(t + m)) \implies \text{req}(t + n)) \]

\[ \forall t. \neg \text{req } t \implies (\exists n. \neg \text{ack}(t + n)) \]

\[ \forall t. \text{ack } t \implies (\forall n. (\forall m. m < n \implies \text{req}(t + m)) \implies \text{ack}(t + n)) \]

\[ \forall t. \neg \text{ack } t \implies (\exists n. \text{req}(t + n)) \]

But this is not very easy to read!
For instance, the assertion,

\[ \forall t. \text{req } t \implies (\forall n. (\forall m. m < n \implies \text{ack}(t + m)) \implies \text{req}(t + n)) \]

...is supposed to say:

"whenever the request signal becomes true, it must remain true until it is acknowledged"

...but it is difficult to translate between this natural language description and the above description in higher-order logic.
A Better Idea!

A much easier notation for expressing constraints of this form is the notation of temporal logic:

\[ \forall t. \text{req } t \implies (\exists n. \text{ack}(t + n)) \]

"every request must eventually be acknowledged"

... where '◊' means "eventually".

(Note: there are no explicit time variables \( t \) in the temporal logic specifications.)
Here is the complete translations into temporal logic notation:

\[(\text{req} \rightarrow (\text{req} \cup \text{ack}))\]

"whenever the request signal becomes true, it must remain true until it is acknowledged"

\[(\text{req} \rightarrow (\Diamond\text{ack}))\]

"every request must eventually be acknowledged"

\[(\text{ack} \rightarrow (\text{ack} \cup (\neg\text{req})))\]

"whenever the acknowledgement signal becomes true, it must remain true until the request signal returns to false"

\[(\text{ack} \rightarrow (\Diamond(\neg\text{req})))\]

"the request signal will eventually return to false after the request is acknowledged"

\[((\neg\text{req}) \rightarrow ((\neg\text{req}) \cup (\neg\text{ack})))\]

"whenever the request signal is false, it will remain false until the acknowledgement signal is also false"
\[ ((\neg \text{req}) \rightarrow (\Diamond (\neg \text{ack}))) \]

"the acknowledgement signal will eventually return to false after the request signal returns to false"

\[ ((\neg \text{ack}) \rightarrow ((\neg \text{ack}) \cup \text{req})) \]

"once false, the acknowledgement signal will remain false until there is a request"

\[ ((\neg \text{ack}) \rightarrow (\Diamond \text{req})) \]

"whenever the acknowledgement signal is false, there will eventually be a request"
To take advantage of this concise notation, the notation of temporal can be embedded in higher-order logic:

We first define the primitive operators,

\[ \vdash_{\text{def}} \Box P = \lambda t. \forall n. P \ (t+n) \]
\[ \vdash_{\text{def}} \Diamond P = \lambda t. \exists n. P \ (t+n) \]
\[ \vdash_{\text{def}} \mathcal{O} P = \lambda t. ((P \ (t+1)) : \text{bool}) \]
\[ \vdash_{\text{def}} P \cup Q = \lambda t. \forall n. (\forall m. m < n \implies \neg (Q \ (t+m))) \implies P \ (t+n) \]
\[ \vdash_{\text{def}} \neg P = \lambda t. \neg (P \ t) \]
\[ \vdash_{\text{def}} P \rightarrow Q = \lambda t. P \ t \implies Q \ t \]
\[ \vdash_{\text{def}} P \wedge Q = \lambda t. P \ t \wedge Q \ t \]
\[ \vdash_{\text{def}} \text{VALID} P = \forall t. P \ t \]

...and derive rules of temporal logic as theorems of higher-order logic:

\[ \forall P \ Q. \ \text{VALID}(P \wedge (\neg Q)) \rightarrow (\ \Box \ (P)) \implies \text{VALID}(P \rightarrow (P \cup Q)) \]

which corresponds to:

\[
\frac{\text{P and (not Q) } \rightarrow \Box P}{P \rightarrow (P \cup Q)}
\]
Exercises:

1. Create a theory for the above temporal logic operators (based on the definitions given). You will have to use some machine-readable symbols instead of □, ◇, ∪, etc.

2. Derive some rules of temporal logic, namely:

∀P Q S. VALID (P → Q) ∧ VALID (Q → S) ⇒ VALID (P → S)

```
expandf (PURE_REWRITE_TAC [ ... ] THEN
  BETA_TAC THEN
  REPEAT_STRIP_TAC THEN
  RES_TAC THEN
  RES_TAC);;
```

∀P Q S.

VALID (P → (◇ Q)) ∧ VALID (Q → S) ⇒
VALID (P → (◇ S))

```
expandf (PURE_REWRITE_TAC [ ... ] THEN
  BETA_TAC THEN
  REPEAT_STRIP_TAC THEN
  RES_THEN (X_CHOOSE_TAC "n:num") THEN
  EXISTS_TAC "n:num" THEN
  RES_TAC);;
```
A low-level circuit model in HOL

by

Brian Andersen
and
Carsten Rickers

HOL Meeting 90
Outline of our talk

- Introduction
- Presentation of the model
- Implementation
- Performing proofs
- Future developments

HOL Meeting 90
Introduction

We need a model that:

- introduce new voltage values
- captures some of the informal arguments used by designers
- introduce resistances and capacitances
- deduce the flow of signals rather than imposing a flow
- is compositional
A compositional model

What is a circuit?

Environment

Circuit

sort:
the set of connection points

Composition:
\( c_1 \circ c_2 \)

Hiding:
\( c \setminus \alpha \)

HOL Meeting 90
A compositional model

The voltage values:

\[ 0 < k_1 < \ldots < k_m < g_1 < \ldots < g_n < \infty \]

The strength order:

HOL Meeting 90
A compositional model

Attributes describing a circuit.

\[ V: \Lambda \rightarrow V \] (the value function)

\[ I: \Lambda \rightarrow V \] (the internal value function)

\[ S: \Lambda \rightarrow S \] (the strength function)

\[ \rightarrow: \Lambda \rightarrow \Lambda \rightarrow \text{bool} \] (the flow relation)

HOL Meeting 90
A compositional model

A static configuration.

Definition:

A static configuration of sort $\Lambda$ is a structure

$$(S, V, I, \rightarrow)$$

which satisfy

- $S(\alpha) = 0 \iff V(\alpha) = Z$
- $I(\alpha) \leq V(\alpha)$
- $\alpha \rightarrow \beta \Rightarrow S(\alpha) \geq S(\beta)$
- $\alpha \rightarrow \beta \Rightarrow I(\alpha) \leq i(\beta) \land V(\alpha) \leq V(\beta)$
- $\alpha \rightarrow \beta \land S(\beta) \in K \cup \{0\} \Rightarrow S(\alpha) = S(\beta)$
- $\alpha \rightarrow \beta \land S(\alpha) = S(\beta) \Rightarrow \beta \rightarrow \alpha$

HOL Meeting 90
A compositional model

Composition of two circuits.

$\sigma_0 = (S_0, V_0, l_0, \rightarrow_0)$ and $\sigma_1 = (S_1, V_1, l_1, \rightarrow_1)$ are static configurations of sort $\Lambda_0$ and $\Lambda_1$ respectively.

$\sigma_0 \circ \sigma_1 = (S, V, l, \rightarrow)$

if $S_0 \sqcup \Lambda_1 = S_1 \sqcup \Lambda_0$ and $V_0 \sqcup \Lambda_1 = V_1 \sqcup \Lambda_0$
and undefined otherwise

$S = S_0 \cup S_1$ and $V = V_0 \cup V_1$ and $\rightarrow = (\rightarrow_0 \cup \rightarrow_1)^*$ and

$l(\alpha) = \sum\{l_0(\beta) \mid \beta \in \Lambda_0 \text{ and } \beta \rightarrow \alpha\} +$

$\sum\{l_1(\beta) \mid \beta \in \Lambda_1 \text{ and } \beta \rightarrow \alpha\}$

for any $\alpha \in \Lambda_0 \cup \Lambda_1$.

Hiding a point in a circuit.

$\sigma = (S, V, l, \rightarrow)$ is a static configuration of sort $\Lambda$.

$\sigma \setminus \alpha = (S \setminus \alpha, V \setminus \alpha, l \setminus \alpha, \rightarrow \setminus \alpha)$

if $\alpha \notin \Lambda$ or $V(\alpha) = l(\alpha) + \sum\{V(\beta) \mid \beta \in \Lambda \setminus \alpha \text{ and } \beta \rightarrow \alpha\}$, and to be undefined otherwise.
A compositional model

A language to describe circuits.

Syntax:

\[
c ::= \text{Pow}(\alpha) \mid \text{Gnd}(\alpha) \mid \text{capkH}(\alpha) \mid \text{capkL}(\alpha) \mid \text{pt}(\alpha) \mid \text{wre}(\alpha) \mid \text{resg}(\alpha,\beta) \mid \text{ntran}(\alpha,\beta,\gamma) \mid \text{ptran}(\alpha,\beta,\gamma) \mid c \circ c \mid c \setminus \alpha
\]

Semantics:

\[
[[\text{ntran}(\alpha,\beta,\gamma)]] = \{ \sigma \in \text{Sta}(\alpha,\beta,\gamma) \mid I(\alpha) = Z \land I(\beta) = Z \land I(\gamma) = Z \land (V(\gamma) = H \iff \alpha \iff \beta) \land (V(\gamma) = L \iff \alpha \parallel \beta) \}
\]

\[
[[c \circ d]] = \{ \sigma \circ \rho \mid \sigma \in [[c]] \text{ and } \rho \in [[d]] \text{ and } \sigma \circ \rho \downarrow \}
\]

HOL Meeting 90
A compositional model

An assertion language.

Syntax:
Value terms: \[ t ::= H \mid L \mid Z \mid X \mid V(\pi) \mid I(\pi) \]

Strength terms: \[ e ::= s \mid S(\pi) \mid e \cdot e \mid e + e \]

Assertions for static configurations:
\[ \phi ::= \pi_0 = \pi_1 \mid \pi_0 \rightarrow \pi_1 \mid t_0 = t_1 \mid t_0 \leq t_1 \mid e_0 = e_1 \mid e_0 \leq e_1 \mid T \mid F \mid \phi \land \phi \mid \phi \lor \phi \mid \neg \phi \mid \exists x. \phi \mid \forall x. \phi \]

Semantics:
\[
[[t_0 = t_1]] = \{ \sigma \in Sta \mid [[t_0]] \sigma \downarrow \text{ and } [[t_1]] \sigma \downarrow \text{ and } \]
\[
[[t_0]] \sigma = [[t_1]] \sigma \}
\]

HOL Meeting 90
We will concentrate on goals as: \([c] \subseteq [A]\)

**compositional model:**

\[
\begin{align*}
\downarrow \text{proof} & & & \\
[c1 \circ c2] & \subseteq [C]
\end{align*}
\]

**Requirements to the implementation:**

- Basic Concepts
- Static Configurations
- Basic Circuits
- Hide, Comp
- Assertion Language
- Tactics

HOL Meeting 90
Basic concepts

points → num in HOL

voltage, strength → new types with operations:

let mosvalue = define_new_type ( mosvalue',
    mosvalue = Z | L | H | Z' );;

let strength = define_new_type ( strength',
    strength = Zero_str | Cap num | Res num | Inf_str' );;

let vlt = new_definition ( vlt',
    "$vlt x y = ~(x = y) \land (x = Z) \lor (y = X')");;

assertion language → build-in operations in HOL +
expressions over the new types

f.x.: (v in = H) ==> (i out = L)

HOL Meeting 90
Static configurations

sort is described by a point function (indirect sort)

static \((n : \text{num} \rightarrow \text{bool})\) \((s : \text{num} \rightarrow \text{strength})\)
\((v : \text{num} \rightarrow \text{mosvalue})\) \((i : \text{num} \rightarrow \text{mosvalue})\)
\((r : \text{num} \rightarrow \text{num} \rightarrow \text{bool})\) =

\((!a. \, n \, a \rightarrow r \, a \, a) \land\)
\((!a \, b \, c. \, n \, a \land n \, b \land n \, c \rightarrow r \, a \, b \land r \, b \, c \rightarrow r \, a \, c) \land\)
\((!a. \, n \, a \rightarrow (s \, a = \text{Zero_str}) = (v \, a = Z)) \land\)
\((!a. \, n \, a \rightarrow (i \, a) \, \text{vle} \, (v \, a)) \land\)
\((!a \, b. \, n \, a \land n \, b \rightarrow r \, a \, b \rightarrow (s \, a) \, \text{sle} \, (s \, b)) \land\)
\((!a \, b. \, n \, a \land n \, b \rightarrow r \, a \, b \rightarrow (i \, a) \, \text{vle} \, (i \, b) \land\)
\hspace{1cm} (v \, a) \, \text{vle} \, (v \, b)) \land\)
\((!a \, b. \, n \, a \land n \, b \rightarrow r \, a \, b \rightarrow\)
\hspace{1cm} (?m. \, s \, b = \text{Cap} \, m) \lor (s \, b = \text{Zero_str}) \rightarrow (s \, a = s \, b)) \land\)
\((!a \, b. \, n \, a \land n \, b \rightarrow r \, a \, b \land (s \, a = s \, b) \rightarrow r \, b \, a)\)

HOL Meeting 90
The basic circuits are defined as predicates:

\[ \text{c n s v i r} \]

let ntrans = new_definition('ntrans',
  "ntrans a b c =
  \n s v i r. static n s v i r \wedge
  (!k. n k --> (k = a) \lor (k = b) \lor (k = c)) \wedge
  \neg(a = b) \land \neg(b = c) \land \neg(a = c) \land
  \neg(b = a) \land \neg(c = b) \land \neg(c = a) \land
  (i a = Z) \land (i b = Z) \land (i c = Z) \land
  \neg(r a c) \land \neg(r c a) \land \neg(r b c) \land \neg(r c b) \land
  ((r ab) \land (r b a) \lor (\neg(r a b) \land \neg(r b a))) \land
  ((v c = H) --> (r a b) \land (r b a)) \land
  ((v c = L) --> \neg(r a b) \land \neg(r b a)) ");;
Composition

\[ \text{comp } c1 \ c2 \ = \ \n s v i r. \]

?n1 s1 v1 i1 1r n2 s2 v2 i2 r2.
(c1 n1 s1 v1 i1 r1) \& (static n1 s1 v1 i1 r1) \&
(c2 n2 s2 v2 i2 r2) \& (static n2 s2 v2 i2 r2) \&
(static n1 s1 v1 i1 r1) \&
(!k. n1 k \& n2 k \implies (s1 k = s2 k) \& (v1 k = v2 k)) \&
(!k. n k = n1 k \lor n2 k) \&
(!k. n k \implies (s k = (n1 k \implies s1 k | s2 k)) \&
(!k. n k \implies (v k = (n1 k \implies v1 k | v2 k)) \&
(!a b. n1 a \& n1 b \& r1 a b = n1 a \& n1 b \& r a b) \&
(!a b. n2 a \& n2 b \& r2 a b = n2 a \& n2 b \& r a b) \&
(!a b c. n1 a \& \neg n2 a \& n1 b \& n2 b \& \neg n1 c \& n2 c \&
\quad r1 a b \& \neg r2 b c = n1 a \& n2 c \& r a c) \&
(!a b c. n2 a \& \neg n1 a \& n1 b \& n2 b \& \neg n2 c \& n1 c \&
\quad r2 a b \& \neg r1 b c = n2 a \& n1 c \& r a c) \&
(!k. n k \implies (i k = vjoinset (\{z. ?j. n j \& r j k \&
\quad ((n1 j \& (z = i1 j)) \lor (n2 j \& (z = i2 j))\})))

HOL Meeting 90
Tactics

An example:

\[ \text{in } s \text{ v i r: comp (inv in x) (inv x out) } \Rightarrow \\
(\text{v in } = \text{H}) \Rightarrow (\text{i out } = \text{H}) \]

can be shown by the properties of \text{inv (+ comp)}

proving methodology:

- Rewrite with the definition of \text{comp}
- Transfer to the assumption list
- Add Facts to the assumption list
- until enough to show the goal

HOL Meeting 90
A proving step

"i out = H" → "i out = H"

!k. n k ==> (v k = (n1 k => v1 k | v2 k))

"v in = H"
"n1 in"
"n in"

|"v1 in = H"

|"n in ==> (v in = (n1 in => v1 in | v in))"

|"v1 in = H"

HOL Meeting 90
A Dynamic Model

Dynamic model as sequence of steady states

\[ \sigma_1 \rightarrow \sigma_2 \rightarrow \sigma_3 \rightarrow \sigma_4 \rightarrow \sigma_5 \]

\[ \uparrow \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow \]

\[ t_1 \quad t_2 \quad t_3 \quad t_4 \quad t_5 \]

time parameter:

\[ ! t. \ (v \ t \ \text{in} = H) \implies (i \ t \ \text{out} = L) \]

let cap = new_definition ('cap',
"cap k a = \n s v i r. (static n s v i r) \land"
(!k. n k = (k = a)) \land
(!t. (Cap k) s le (s t a)) \land
(!t. (s (t+1) a = Cap k) \implies (i (t+1) a = v t a)) \land
(!t. ((Cap k) s le (s t a) \implies (i t a = Z)))");;

HOL Meeting 90
Future developments

We have implemented a hardware model in HOL that works reasonable

Still some work to do

Better transistor description can be obtained

Extensions to other types of proves as: $[c1] = [c2]$

Can easily be extended to a dynamic model

HOL Meeting 90
THE VERIFICATION OF A
PARAMETERIZED MEAD&CONWAY
ALU CORE IN HOL

CATIA ANGELO
LUC CLAESSEN
HUGO DE MAN

imec vzw

MAY 90
OUTLINE

- PRELIMINARY CONSIDERATIONS

- INTRODUCTION

- THE DESIGN PROCESS
  1- LOGICAL FUNCTIONS
  2- ARITHMETIC FUNCTIONS

- THE PROOF
  1- USING THE KNOWLEDGE OF THE DESIGN PROCESS
  2- STRATEGY AND PROOF FLEXIBILITY
  3- GOING FROM A IRREGULAR STRUCTURE TO A REGULAR STRUCTURE

- CONCLUSIONS
1- GLOBAL GOALS:

- LEARN HOL;

- DEVELOP A METHODOLOGY FOR HARDWARE VERIFICATION USING HOL;

- IDENTIFY THE BOTTLENECKS OF THE METHODOLOGY AND HOL;

- LIBRARY OF PRE-PROVEN MODULES;

- LIBRARY OF USEFUL STRATEGIES TO PROVE HARDWARE CORRECTNESS (TACTICS AND TACTICALS).

2- LOCAL GOAL: ALU VERIFICATION IN HOL

THE ALU CORE IS A GOOD EXAMPLE BECAUSE THE BEHAVIOR IS SPREAD ALL OVER THE DATA FLOW AND IT IS NOT COMPLETELY STRUCTURED IN SUB-BEHAVIORS.
PRELIMINARY CONSIDERATIONS

1- GLOBAL GOAL:
   - DEVELOP A METHODOLOGY FOR HARDWARE VERIFICATION USING HOL AND BOYER MOORE IN SUITABLE LEVELS
   - IDENTIFY THE BOTTLENECKS OF:
     - THE METHODOLOGY
     - HOL
     - BOYER MOORE

   - HOL:
     - LIBRARY OF PRE-PROVEN MODULES
     - LIBRARY OF USEFUL STRATEGIES TO PROVE HARDWARE CORRECTNESS (TACTICS AND TACTICALS)

2- LOCAL GOAL: ALU VERIFICATION IN HOL
   THE ALU CORE IS A GOOD EXAMPLE BECAUSE THE BEHAVIOR IS SPREAD ALL OVER THE DATA FLOW AND IT IS NOT COMPLETELY STRUCTURED IN SUB-BEHAVIORS
INTRODUCTION
- THE GOAL IS TO PROVE THAT:
  ALU IMPLEMENTATION = ALU SPECIFICATION
- THE ALU IS PARAMETERIZED ON THE WORD SIZE N

- THE ALU SHOULD PERFORM LOGICAL OR ARITHMETIC OPERATIONS ON THE VECTORS "A" AND "B" DEPENDING ON THE CONTROLS. THE SPECIFICATION IS MADE IN A PROGRAMMING LANGUAGE LIKE.

- THE VERIFICATION LEVELS:
1. LOGICAL FUNCTIONS:

<table>
<thead>
<tr>
<th>DECIMAL CODE</th>
<th>BINARY CODE CT3 CT2 CT1 CT0</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 1</td>
<td>(A ∧ B)'</td>
</tr>
<tr>
<td>2</td>
<td>0 0 1 0</td>
<td>(A' ∨ B)</td>
</tr>
<tr>
<td>3</td>
<td>0 0 1 1</td>
<td>A'</td>
</tr>
<tr>
<td>4</td>
<td>0 1 0 0</td>
<td>(A ∨ B)</td>
</tr>
<tr>
<td>5</td>
<td>0 1 0 1</td>
<td>(A ⊕ B)</td>
</tr>
<tr>
<td>6</td>
<td>0 1 1 0</td>
<td>B</td>
</tr>
<tr>
<td>7</td>
<td>0 1 1 1</td>
<td>(A' ∧ B)</td>
</tr>
<tr>
<td>8</td>
<td>1 0 0 0</td>
<td>(A ∨ B')</td>
</tr>
<tr>
<td>9</td>
<td>1 0 0 1</td>
<td>B'</td>
</tr>
<tr>
<td>10</td>
<td>1 0 1 0</td>
<td>(A ⊕ B)'</td>
</tr>
<tr>
<td>11</td>
<td>1 0 1 1</td>
<td>(A ∨ B')</td>
</tr>
<tr>
<td>12</td>
<td>1 1 0 0</td>
<td>A</td>
</tr>
<tr>
<td>13</td>
<td>1 1 0 1</td>
<td>(A ∧ B')</td>
</tr>
<tr>
<td>14</td>
<td>1 1 1 0</td>
<td>(A ∧ B)</td>
</tr>
<tr>
<td>15</td>
<td>1 1 1 1</td>
<td>0</td>
</tr>
</tbody>
</table>
GFB CIRCUIT
THE GFB BAR

CT3  CT2  CT1  CT0

A0 → G F B → OUT0
B0 →

A1 → G F B → OUT1
B1 →

A2 → G F B → OUT2
B2 →

...

An-1 → G F B → OUTn-1
Bn-1 →

CT3  CT2  CT1  CT0

A[(n-1)...0] → G F B → OUT[(n-1)...0]
B[(n-1)...0] →

THE GFB BAR PERFORMS ALL LOGICAL OPERATIONS
WITH TWO VECTORS A AND B.
2- ARITHMETIC FUNCTIONS

THEY ARE ALL A KIND OF ADDITION

<table>
<thead>
<tr>
<th>C</th>
<th>A</th>
<th>B</th>
<th>K</th>
<th>P</th>
<th>SUM</th>
<th>CARRY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ K = A \]

\[ P = \text{EXOR} (A, B) \]

\[ \text{CARRY} = (P \land C) \lor (P' \land K) \]

\[ \text{SUM} = \text{EXOR}(P, C) \]

FOR N BITS:

\[ K_j = A_j \quad \Rightarrow \quad \text{GFB} \ (A_j, B_j) \]

\[ P_j = \text{EXOR} (A_j, B_j) \quad \Rightarrow \quad \text{GFB} \ (A_j, B_j) \]

\[ C_j = (P_j \land C_{j-1}) \lor (P_j' \land K_j) \quad \Rightarrow \quad \text{CARRY} \ (P_j, K_j, C_{j-1}) \]

\[ S_j = \text{EXOR} (P_j, C_{j-1}) \quad \Rightarrow \quad \text{GFB} \ (P_j, C_{j-1}) \]

\[ K_v, P_v, S_v \quad \Rightarrow \quad 3 \ \text{GFB BARS} \]

\[ C_v \quad \Rightarrow \quad \text{MUX CHAIN} \]
IDEAL CARRY CHAIN

BUT...

"CORRECT" INPUTS
INVERTED OUTPUTS

HOW TO CASCADE?
POSSIBLE SOLUTIONS:

1)

BAD SOLUTION!
WASTE OF TIME AND AREA!
IF INSTEAD OF $K_j = A_j$ ONE GENERATES $K_j = A_j'$ IN THE K GFB BAR:

SINCE BUFFERING CIN AND COUT IS DESIRED THE CHAIN IS SHIFTED
'GFBB' BEHAVES LIKE GFBB BUT IT HAS A DIFFERENT HARDWARE
<table>
<thead>
<tr>
<th>DECIMAL CODE</th>
<th>BINARY CODE</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 1</td>
<td>(A' ∨ B)</td>
</tr>
<tr>
<td>2</td>
<td>0 0 1 0</td>
<td>(A ∧ B)'</td>
</tr>
<tr>
<td>3</td>
<td>0 0 1 1</td>
<td>A'</td>
</tr>
<tr>
<td>4</td>
<td>0 1 0 0</td>
<td>(A ∨ B)'</td>
</tr>
<tr>
<td>5</td>
<td>0 1 0 1</td>
<td>(A ⊕ B)'</td>
</tr>
<tr>
<td>6</td>
<td>0 1 1 0</td>
<td>B'</td>
</tr>
<tr>
<td>7</td>
<td>0 1 1 1</td>
<td>(A ∨ B)'</td>
</tr>
<tr>
<td>8</td>
<td>1 0 0 0</td>
<td>(A ∨ B)</td>
</tr>
<tr>
<td>9</td>
<td>1 0 0 1</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>1 0 1 0</td>
<td>(A ⊕ B)</td>
</tr>
<tr>
<td>11</td>
<td>1 0 1 1</td>
<td>(A' ∧ B)</td>
</tr>
<tr>
<td>12</td>
<td>1 1 0 0</td>
<td>A</td>
</tr>
<tr>
<td>13</td>
<td>1 1 0 1</td>
<td>(A ∧ B)</td>
</tr>
<tr>
<td>14</td>
<td>1 1 1 0</td>
<td>(A ∧ B')</td>
</tr>
<tr>
<td>15</td>
<td>1 1 1 1</td>
<td>0</td>
</tr>
</tbody>
</table>
THE CARRY CHAIN CELLS

ODD CELL

EVEN CELL
LOGICAL OPERATIONS:
- THE OPERATION IS PERFORMED IN THE P BAR
- CR IS SUCH THAT OUT[j] = P[j]
- WHAT HAPPENS IN THE K BAR AND IN THE CARRY CHAIN DOES NOT MATTER
- THE DATA FLOW IS HORIZONTAL

ARITHMETIC OPERATIONS:
- THEY CAN BE SEEN AS A KIND OF ADDITION
- THE DATA FLOW IS BOTH HORIZONTAL AND VERTICAL
- HORIZONTAL FLOW: PARALLEL
- VERTICAL FLOW: SERIAL (SPEED BOTTLENECK)
IMPROVING THE CARRY CHAIN SPEED

\[ P = 1 \text{ MEANS: CARRY INFORMATION IS PROPAGATED} \]
\[ (P_i = 1) \land (P_{i+1} = 1) \land (P_{i+2} = 1) \land (P_{i+3} = 1) \]
\[ \implies c_{i+3} = c_{i-1} \]

E: EVEN CELL;  O: ODD CELL;  B: BYPASS CELL

E O E O B E O E O B E O E O B ... 

REMARK: THE CHAIN COULD BE FASTER ELIMINATING THE OUTPUT INVERSION OF THE BYPASS CELL AND CHANGING ITS STRUCTURE TO:

E O E O B O E O E B E O E O B ...
THE BYPASS CELL

Pi
Pi+1
Pi+2
Pi+3

S' → S

ci+3'

ci-1'

SEL

S

ci+3  better'

VDD

Pi
Pi+1
Pi+2
Pi+3

GND

S' → S

VDD

S

ci-1'

S

GND

ci+3'

S'
1 - Using the knowledge of the design process we have a hint about what to prove or consider like:

- The GFB performs the logical operations
- GFBB (CT, A, B) = GFB (CT, A, B')
- Carry chain with bypass = Carry chain without bypass cells
- Carry chain + result bar perform some ideal operations
- When executing a logical function:
  ALU behavior = P bar behavior
- The arithmetic operations are a kind of addition

2 - Strategy and proof flexibility
CONSIDER IN THE ALU THAT:

- SOMEONE USES A BYPASS OF SIZE DIFFERENT OF FOUR AND/OR
- SOMEONE CHANGES THE START POSITION OF THE BYPASS CELL AND/OR
- SOMEONE OMITS THE OUTPUT INVERTER IN THE BYPASS CELL AND CHANGES THE STRUCTURE OF THE CARRY CHAIN

GOING FROM IMPLEMENTATION TO IMPLEMENTATION' WOULD

- REQUIRE AN ADDITIONAL STEP
- NOT REQUIRE RE DOING ALL THE VERIFICATION PROCESS

3- GOING FROM A IRREGULAR STRUCTURE TO A REGULAR STRUCTURE:

- THE BYPASS CELL ELIMINATION
- THE CARRY CHAIN AND THE RESULT BAR TRANSFORMATIONS
- THE FOUR BAR TRANSFORMATIONS
BYPASS CELL ELIMINATION

\[ \text{cin'} = \text{cin}_\text{bet'} \implies \]

\begin{align*}
\text{cin}_\text{bet'} & \quad \text{cin'} \\
\text{o}_\text{cin'} & \quad \text{o}_\text{c0} & \quad \text{o}_\text{c1'} & \quad \text{o}_\text{c2}
\end{align*}

\[ = \]

\begin{align*}
\text{P}[3...0] & \quad \text{K}[3...0] \\
\text{cin}_\text{bet'} & \quad \text{cout}_\text{bet'} \\
\text{cout'} & \quad \text{cout}_\text{bet'} \\
\text{o}_\text{cin'} & \quad \text{o}_\text{c0} & \quad \text{o}_\text{c1'} & \quad \text{o}_\text{c2}
\end{align*}
CARRY CHAIN TRANSFORMATIONS

\[
\begin{align*}
\text{EOEO} & \rightarrow \text{BYPASS BLOCK} \rightarrow \text{BYPASS BLOCK} \rightarrow \text{BYPASS BLOCK} \\
\cdots & \rightarrow \text{BYPASS BLOCK} \rightarrow \text{BYPASS BLOCK} \rightarrow \text{EOEO} \\
= & \quad \text{EOEO} \rightarrow \text{EOEO} \rightarrow \cdots \rightarrow \text{EOEO} \\
= & \quad E \rightarrow O \rightarrow E \rightarrow O \rightarrow \cdots \rightarrow O
\end{align*}
\]
IRREGULAR ALU = REGULAR ALU
REGULAR ALU = SPECIFICATION
IRREGULAR ALU = SPECIFICATION

REGULAR ALU = SPECIFICATION ==> SLICE REASONING AND INDUCTION
CONCLUSIONS:

1- THE REASONING USED IN THE HARDWARE DESIGN SHOULD BE USED IN THE VERIFICATION

2- REMOVING IRREGULARITIES FROM THE IMPLEMENTATION BEFORE GOING TO THE SPECIFICATION LEVEL IS A POWERFUL STRATEGY TO STRUCTURE THE CORRECTNESS PROOF
The HOL Verification of ELLA Designs

Richard Boulton, Mike Gordon, John Herbert, John Van Tassel

University of Cambridge Computer Laboratory
New Museums Site
Pembroke Street
Cambridge CB2 3QG
England

rjb@cl.cam.ac.uk
Phone: +44 223 334729
Fax: +44 223 334678
Motivation

Two methods used to specify hardware:

- Hardware description languages, e.g. ELLA, VHDL
- Formal systems, e.g. HOL, Boyer-Moore logic

There is currently a gap between these two methods.
Our Aims

To build a bridge between the two methods for ELLA and the HOL proof assistant.

- Make formal specification and proof available within a conventional design process

- Provide access to the engineering tools in a theorem proving environment

Our Approach

To semantically embed a subset of ELLA in higher order logic.
Semantic Embedding

Each construct in the chosen subset of ELLA is represented by a logical constant.

The constants are defined to have the behaviour of the corresponding language construct.

The use of 'semantic constants' gives us:

- A simple translation from ELLA to HOL terms

- The ability to pretty-print the HOL terms as the original ELLA constructs

The semantic constants can be expanded and simplified to yield a more conventional logical specification of the hardware.

The behavioural specification can be given as high-level ELLA or in pure logic.
The System

ELLA TEXT

Parser

Parse Tree

Compute Semantics

error

HOL TERM

Pretty-Printer

ELLA IN HOL

= Program

= Data
An Example

A simple circuit:

\[ \text{in} \quad \underline{\text{INV}} \quad \text{out} \]

The ELLA for the circuit:

\begin{verbatim}
BEGIN
  MAKE INV: out.
  JOIN in -> out.
OUTPUT out
END
\end{verbatim}

The HOL representation of the ELLA:

\begin{verbatim}
εOUTPUT.
∃out out_fn.
SERIES
  [MAKE[[MAKEITEM INV out_fn]];
   JOIN
     [JOINITEM in out out_fn(λx'. εf1'. ∀t'. f1' t' = x' t')]]
OUTPUT
out
\end{verbatim}
Eliminating Semantic Constants

Expanding

\[ \text{\$OUTPUT.} \]
\[ \exists \text{out out_fn.} \]
\[ \text{SERIES} \]
\[ \text{MAKE}\left[\text{MAKEITEM INV out_fn}\right]; \]
\[ \text{JOIN} \]
\[ \text{JOINITEM in out out_fn}\left(\lambda x'. \epsilon f_1'. \forall t'. f_1'\ t' = x'\ t'\right)\] \]
\[ \text{OUTPUT} \]
\[ \text{out} \]

with the definitions

\[ \text{MAKEITEM} (\text{dev:*}) \text{ fun} = (\text{fun} = \text{dev}) \]
\[ \text{JOINITEM in} (\text{out:**}) \text{ fun} (g:*\rightarrow**) = ((\text{fun} (g \text{ in})) = \text{out}) \]

and simplifying the type-casting function, yields

\[ \text{\$OUTPUT.} \]
\[ \exists \text{out out_fn.} \]
\[ \text{SERIES} \]
\[ \text{MAKE}\left[\text{out_fn = INV}\right]; \]
\[ \text{JOIN}\left[\text{out_fn } \left((\lambda x'. x') \text{ in}\right) = \text{out}\right] \]
\[ \text{OUTPUT} \]
\[ \text{out} \]
Eliminating Semantic Constants

Expanding

εOUTPUT.
∃out out_fn.
SERIES
[MAKE[[out_fn = INV]];
 JOIN[[out_fn ((λx'. x') in)) = out]]
OUTPUT
out

with the definitions

MAKE 1 = (ITLIST $\& (MAP (λx. ITLIST $\& x T) 1) T)
JOIN 1 = (ITLIST $\& 1 T)
(SERIES [] output (unit:*)) = (output = unit)) ∧
(SERIES (CONS x 1) output unit =
(x ∧ (SERIES 1 output unit)))

and simplifying, yields

εOUTPUT.
∃out out_fn.
(out_fn = INV) ∧
(out_fn in = out) ∧
(OUTPUT = out)
Simplification

$$\varepsilon_{\text{OUTPUT}}.$$  
$$\exists \text{out} \text{ out}_\text{fn}.$$  
$$(\text{out}_\text{fn} = \text{INV}) \land$$  
$$(\text{out}_\text{fn \ in} = \text{out}) \land$$  
$$(\text{OUTPUT} = \text{out})$$

simplifies to

$$\varepsilon_{\text{OUTPUT}}.$$  
$$\exists \text{out}.$$  
$$(\text{INV \ in} = \text{out}) \land$$  
$$(\text{OUTPUT} = \text{out})$$

which simplifies to

$$\varepsilon_{\text{OUTPUT}}. \text{ OUTPUT} = \text{INV \ in}$$

which simplifies to

INV \ in
ELLA constants

ELLA constants are used in two ways:

- As initial values for devices with state
- As 'patterns' in the CASE statement

There is only one syntactic category for constants. A given constant may make sense in only one of the two roles above.

Context-free translation of constants.

Constants are modelled as predicates. This is consistent with their use as patterns.

For initial values, Hilbert's $\varepsilon$-operator is used.

There is a problem with the semantics of constants when used as patterns ...
The ELLA unspecified value and constants

ELLA features an unspecified value denoted by ?.

? may be generated:

- by arithmetic operations producing a result out of range
- by CASE statements with choosers which are not exhaustive
- explicitly

? represents one of the possible values of the appropriate type, but it is not known which.
The ELLA unspecified value and constants

An unknown value of some type can readily be represented in higher order logic.

However, there is one place in the semantics of ELLA where \( ? \) behaves as an additional value of the type. This is not so readily represented in HOL.

For an ELLA type \texttt{bit} with the two possible values \texttt{hi} and \texttt{lo}
\[
\texttt{hi} \mid \texttt{lo}
\]

is a constant which 'matches' \texttt{hi} or \texttt{lo}, but \textit{not} \( ? \).

\texttt{bit}

is a constant which 'matches' \texttt{hi} or \texttt{lo} or \( ? \).
Lifting

We represent an additional value of an ELLA type by using a lifted HOL type.

The ELLA type bit is represented by the HOL type (bit)lifted.

The type constructor lifted is defined by the HOL type definition

define_type 'lifted_Axiom' 'lifted = UU | LIFT *'

So, the ELLA value hi is represented by "LIFT hi".

lo is represented by "LIFT lo".

The unspecified value of type bit is represented by "UU:(bit)lifted".

Uniform approach.

Lifted Boolean values are also used.
Conclusions and Future Work

- A semantics has been given for a substantial subset of the ELLA language, and a translator produced for the subset.

- The translation is simple. There is a straightforward relationship between ELLA syntax and its semantics in HOL.

- Parser and pretty-printer support has been provided.

- Lifting causes difficulties. New version of semantics without lifting and not quite conforming to the semantics of ELLA?

- Subset chosen is large. Discard lightly used parts?

- Research is required to decide whether we have a useful subset.


- Design and verification case studies: UART, small microprocessor.

- Develop a methodology for using the combined HOL/ELLA system.
HOL Semantics of SILAGE

Presentation at the Third HOL User Meeting

Ton Kalker
Philips Research Laboratories Eindhoven
P.O. Box 80000
5600 JA Eindhoven
The Netherlands
1. Introduction
2. SILAGE
3. Semantics: basics
4. Semantics: types
5. Semantics: basic types
6. Semantics: array types
7. Semantics: proposals
8. Example
9. Simultaneous induction
10. Conclusions
Introduction

- **Silicon compiler**: $(\text{Program} \rightarrow \text{Silicon})$

- **SILAGE**: input language for CATHEDRAL (PIRAMID) silicon compiler for DSP applications

- **SILAGE program $P^{\text{compilation}}$ architecture**:
  
  1. **EXU’s** ($\text{ALU}$, $\text{RAM}$, $\text{ROM}$),

  2. **busses** connecting the EXU’s and a

  3. **controller**
Introduction

- **SILAGE** ≡ linear representation of finite set $S = \{G_1, \ldots, G_n\}$ of signal flow graphs

- SILAGE programs denote *stream* transformers

- stream ≡ infinite vector in time ($\sim \mathbb{N}$)

- Compiler determines absolute time-scale

- behavior ≡ abstraction from absolute time-scale
Introduction

- Wanted:
  - method to reason about behavior (≡ semantics)
  - method to compare SILAGE programs with mathematical formulae

- Solution:
  - mathematics ← HOL-logic
  - express semantics in HOL-logic

- Spin-off:
  - discovery of bad language design
  - proposals for improvement of SILAGE
Sample SILAGE program:

```
FUNC main (in:NUM<3,0>) out:NUM<3,0> =
    NUM<3,0> : tmp;
BEGIN
    out = help(tmp).out;
    tmp = in@1;
END;

FUNC help (in:NUM<3,0>) out:NUM<3,0> =
BEGIN
    out = in + in@1;
END;
```
• $\text{Time} \equiv \mathbb{T} \overset{\text{def}}{=} \mathbb{N}$

• $\text{Stream} \equiv (\alpha)^{\text{str}} \overset{\text{def}}{=} \mathbb{T} \rightarrow \alpha$

• $\text{Delay} (\@) : \Delta : (\alpha)^{\text{str}} \rightarrow (\alpha)^{\text{str}}$

\[ \forall \text{nat } t. \]
\[ (t \geq n) \Rightarrow \]
\[ \Delta \text{nat } t = a(t - n) \]

$\Delta \text{na} \ (\text{semantically) initially undefined}!!!!$
• SILAGE type:
  basic type & array structure
  \sim
  \text{NUM}<w,d>\ [m1] [m2] \ldots [mn]

• Basic type: \text{NUM}<w,d>:
  \sim
  array of \ w \ booleans with \ d < w \ booleans after the decimal point.

• HOL-logic lacks parametrized types
  \Rightarrow
  SILAGE types \sim \textit{predicates on a fixed HOL-type}
a: NUM<w,d> is translated to

- a term a of HOL type (B × (B)list × (B)list)str accompanied by

- the predicate

\[ \forall t. \text{let } (s, i, f) = (a \ t) \text{ in } \]
\[ (\text{LENGTH } i = w - d - 1) \land \]
\[ (\text{LENGTH } f = d) \]

Suites for shift and bitwise operations!!
Possibility 2

\(a: \text{NUM}[^w, d]\) is translated to

- a term \(a\) of HOL type \((\mathbb{Q})^{str}\) accompanied by

- the predicate

\[
\forall t. \text{let } b = (a \cdot t) \times 2^{d+1} \text{ in } \\
\text{let } c = (2^w) \text{ in } \\
(-c \leq b) \land (b < c)
\]

Suited for arithmetic operations!!
• *if-then-else* construct

```
  IF b:NUM<1,0> -> e1 || e2 FI
```

NUM<1,0> serves as the type **B**!!

• **THE SILAGE TYPE SYSTEM DOES NOT MAKE EXPLICIT THE DIFFERENCE BETWEEN LOGICALLY DISTINCT TYPES!!**

• **Recommendation:** Introduction of a more complex type system in SILAGE.
Array types

a: \texttt{baty}[N_1] is most easily interpreted

- a term $a$ of HOL type $((\texttt{baty})\texttt{list})\texttt{str}$ accompanied by

- the predicates

\[ \forall t. \text{LENGTH}(a t) = N_1 \]

\[ \forall t n. (n < N_1) \Rightarrow (P(EL\ n(a t))) \]
Evaluation scheme of array types in SILAGE conflicts with with evaluation scheme of list types in HOL

```
#define SYMB NUM<1>[8]  % abbreviation %
.
SYMB[16] : tmp  % declaration %
.
tmp[12] = ....  % indexing %
```

Highest index allowed in tmp is 7!!
• Initialization:

1. compiler independence

2. no default values available (types)

```c
BOOL[4] : tmp  % declaration %

(j : 0 .. 3) ::
    (k : 0 .. 2) ::
        tmp[j][k] = j*k;
```
Basic types:

1. \text{BOOL} \sim B

2. \text{NUM<}w,d> \sim \mathbb{Q}+

3. \text{INT<}w,d> \sim \mathbb{Q}

4. \text{COMPLEX<}w,d> \sim \mathbb{C}_{\mathbb{Q}}

Functions with empty parameter list.

1. a \text{CONST} construct: \textit{streams constant in time}

2. the possibility of defining functions with empty parameter list
CONST sinus : INT<13,12>[12] = 
    sinus = [0;.....];

FUNC wobble() : INT<13,12> =
    NUM<4,0> : count;
BEGIN
    count{0} = 4;
    count = IF (count < 11) -> count@1 + 1
    ||
    0
    FI;
    return = sinus[count];
END;
Simultaneous induction

\[
\begin{align*}
\text{tmp1} &= \text{tmp2} + \text{tmp1@1} \\
\text{tmp2} &= \text{tmp1} + \text{tmp2@1}
\end{align*}
\]

translates to

\[
\begin{align*}
\text{tmp1}(n + 1) &= (\text{tmp2}(n + 1)) + (\text{tmp1} n) \\
\text{tmp2}(n + 1) &= (\text{tmp1}(n + 1)) + (\text{tmp2} n)
\end{align*}
\]

HOL needs induction mechanism which:

- fails on this example and

- succeeds on acyclic induction schemes
Conclusions

- Language design can benefit from mathematical methods

- Extension SILAGE+ suited for compilation as well as verification

- Automatic translation of SILAGE+ into HOL-logic necessitates a better natural induction scheme