Abstracts from the 1989 HOL Users Group Meeting

Phillip J. Windley
(Editor)

Division of Computer Science
Department of Electrical Engineering and Computer Science
University of California, Davis

December 14-15, 1989

Editor's Note

The 1989 HOL User's Group Meeting was held December 14-15, 1989 at Trinity Hall College, Cambridge University. This document gives abstracts, prepared individually by the speakers, of the technical presentations from the conference. For more information on a given talk, please contact the authors directly via electronic mail; their addresses are provided.

Keynote Talk: Formal Verification: The Art of Being Economical with the Truth

Keith Hanna
fkh@ukc.ac.uk
University of Kent, UK

Design verification can be seen as a game played between society, wishing to protect itself from disaster, and designers, wishing to promote their products. Society's interests are entrusted to an authority whose task is to select a verification method and oversee its application. A verification method has two components: a formal logic and an interpretation. The purpose of the former component is (to borrow a phrase) to be "economical with the truth;" its axioms and rules of inference should be few in number and self-evidently sound. Higher-order logic can adequately meet this requirement. The purpose of the latter component, the interpretation, is to relate the formal logic to the physical world. It is vitally important that this relationship be examined with full rigor; the meaning of each primitive symbol must be precisely defined, the truth of each axiom and the intent of each definition must be checked. This activity lies at the intersection of engineering and logic and thus responsibility (between designer and logician) for undertaking it can be blurred. This in turn can lead to a situation where the use of formal methods actually decreases design integrity; the logician verifies the design against simplistic (i.e., false) assumptions and the engineer, knowing that the design will be "formally verified," lets drop his guard.

A criterion sometimes advanced for formal verification is "Would I fly it?." This is a rather weak test to apply since experience tells us that most designs do, in fact, work satisfactorily most of the time. A stronger criterion for formal verification would be game-based, perhaps along the following lines: "Would I entrust myself to an apparatus designed and verified (albeit strictly in accordance with the rules of the verification method), not by a benign engineer, but rather by an omniscient demon setting out, with malicious intent, to exploit every weakness in the formal logic, fragility in its computational implementation and imprecision in its interpretation?."

The second part of the talk described *VERITAS+*, a design logic having dependent types and subtypes [1]. Such types are useful for describing the bounded entities (e.g., n-bit numerals) typically occurring in digital system specifications. For example, the function `val` that maps a numeral to a number may be given the dependent type:

```
val: [b:natp] * [l:nat] -> V(b,l) -> N (b exp l)
```

(Read as: `val` takes a positive natural number b and a natural number l and then takes a length-l base-b numeral, and yields a natural number in the subrange less than b to the power of l.) The penalty incurred in the use of dependent types is the loss of decidable type-checking; in practice, a tactic-based approach allows a reasonable degree of automation to be realized. The main advantage of dependent types is the increased clarity of specifications.

References:

1. "Specification and Verification using Dependent Types," Technical Report, University of Kent, 1989.

1 Software and Protocol Verification

Session Chair: Roger Hale, Cambridge University

December 14, Afternoon

Speakers:

John Cullyer	*Railway Signalling*
Rachel Cardell-Oliver	*Specifying and Verifying a Class of Protocols*
Albert Cammilleri	*Mechanizing CSP Trace Theory in HOL*
Joakim von Wright	*Formalizing Program Refinements*

Railway Signalling

John Cullyer
eswjc@warwick.ac.uk
Warwick Univeristy, UK

Abstract not available

Ideas for Specifying a Class of Protocols and Verifying Implementations using Higher Order Logic

Rachel Cardell-Oliver
rco%uk.ac.cam.cl@nsfnet-relay.ac.uk
Cambridge Univeristy and
Defense Science and Technology Organization, Australia

Computer network protocols are members of the class of real time concurrent programs. The class of protocols which, say, transfers data from one place to another in a computer network share not only the same function, but also the same behavior in the sense of the basic mechanisms which all these protocols use.

The most obvious similarity between protocols of the same class is their physical structure. Each protocol is effected by a sender and receiver (programs on separate computers) which communicate, only, via a bi-directional channel. Protocol structure is captured in HOL using techniques developed for hardware specification. Each physical entity (sender, receiver and channel) is modeled by a higher order predicate and communication between the entities is modeled by shared parameters to these predicates.

Less obvious to model is the mechanism for reliable communication over an unreliable channel: positive acknowledgment. Each data message which is transmitted must be acknowledged by a returned message. Transmission and reception predicates in the sender and receiver achieve this by continuously transmitting data or acknowledgments (respectively) and using local state variables to chart progress.

Properties which vary between protocols of the same class are captured by parameters of the specification. For example, window sizes and channel delays are modeled by parameters of type *sequence* and *time* respectively. The choice of which data message to send next is modeled by a parameter of type $time \rightarrow sequence$ which, at any time, chooses the next message to transmit.

The correctness of a protocol depends on complex real time interactions between the sender and receiver programs and the channel. The channel's response is non-deterministic. For example, messages may be delivered or lost and if delivered may be delayed for a variable, but bounded, time. Real time intervals are specified using a constant added or subtracted from the current time. For example a timeout occurs at time t if the value of the sender's state at time t is the same as its state at time $t - TIMEOUT$. Protocol liveness is determined by an interval of maximum persistence, $maxP$, in the sender. That is, the protocol will be aborted if the sender's state at time t is the same as its state at time $t - maxP$.

Verification that the protocol behavior specification discussed above satisfies the functional specification for a class is still in progress but near completion. Earlier specifications are being restructured and the proofs updated. The next task is to specify and verify real time protocol implementations. The following is a possible route for performing this task.

1. Define mappings between the names of program variables in an implementation and the variables of the HOL specification.

2. Define the relationship between the real time scales of the program and the abstract time scales of the HOL specification. Techniques for doing this can be found in the work of Tom Melham, Jeff Joyce, John Herbert.

3. Show, possibly using Floyd-Hoare proof rules, that the updated values of program variables correspond to the updated variables of the HOL specification. A HOL description of a Floyd-Hoare proof system similar to the one defined by Mike Gordon could be used for this task.

4. Interpret real time events such as packet arrival and timeout. Should these be modeled using polling, a wait statement which jumps to first condition to be satisfied, a flag held high until the event is recognized etc.?

Finally, having shown that a particular implementation is a member of the class of data transfer protocols, I would like to prove that it is a non-trivial member of that class by proving properties about the protocol's performance. One option for doing this is to assign probabilities to the non-deterministic choices of the channel and from this deduce real time performance properties of the protocol.

Mechanizing CSP Trace Theory in HOL

Albert Cammilleri
ac@acamille@hplb.hpl.hp.com
Hewlett Packard Laboratories, UK

In this talk I describe a mechanization of the formal language CSP in higher order logic. CSP is one of several process algebras developed for reasoning about concurrency and communication, and mathematical proof plays a major part in reasoning using the language.

The CSP semantics modeled in HOL is that of traces. This is one of the simplest semantic models developed for CSP over the years, but is nonetheless useful and provides insight into the nature of the problems involved in mechanizing the process algebra, and the practicality of the final results.

One thing which immediately becomes evident from the mechanization is the amount of general mathematics required for reasoning in the semantics of CSP (e.g. set theory, fixed points). The talk describes the mechanization of some fundamental CSP notions such as events, alphabets and traces using the necessary mathematical foundations, and how a data type is defined to represent CSP processes. It is shown how the trace semantics of the CSP operators are defined conservatively on this data type, and how high level process algebra laws are derived from the definitions of the operators. Finally it is shown how the adopted semantics can be separated from the syntax of the language by first defining a syntactic type of CSP processes using Melham's type definition package, and then formalizing a denotational semantics for the language using the trace semantics.

Formalizing Program Refinements

Joakim von Wright
jwright@finabo.abo.fi
Abo Academi, Finland

The refinement calculus is a theory of correctness preserving program transformations, based on based on Dijkstra's weakest precondition. It has proved useful for different kinds of program development, including data refinement and development of parallel programs. We write $S <= S'$ (saying that S is refined by S') if $wp(S, Q) => wp(S', Q)$ holds for all predicates Q. Thus the relation $<=$ preserves total correctness.

Program development in the refinement calculus starts from a specification S_0, and through a series

$$S_0 <= S_1 <= ... <= S_n$$

of transformations, ends with a program S_n.

Our goal using HOL is threefold. First, we want to formalize refinement concepts in HOL (i.e., define a specification language with weakest precondition semantics). Second, we want to check the proofs of refinement laws and transformation rules. Third, we want to do program refinements within the HOL theory of refinement, program development resulting in a theorem $\vdash\ S_0\ <=\ S_n$ within the refinement theory. Working within the same theory where the transformation rules were proved correct guarantees that the basis for the work is solid.

We define predicates semantically, as the type $state \rightarrow bool$, where state is the type $var \rightarrow val$. The variables could be strings and the values natural numbers. Operators on predicates are defined by lifting, e.g. (p and q) is defined as

$$\lambda\ (s : state)\ .\ s\ p \wedge s\ q$$

Expressions have type $state \rightarrow val$. Substitutions on predicates are a bit tricky to define, since predicates are not syntactically defined. Predicate transformers have type $pred \rightarrow pred$. Among other things we prove that every monotonic predicate transformer has a least fixpoint.

Commands are a recursive type *cmd*. Our language is Dijkstra's guarded commands, extended with assertions, nondeterministic assignment and block with local variables. The semantics is given by a function $wp : cmd \rightarrow pred \rightarrow pred$, associating every command with its weakest precondition predicate transformer. For iteration we give a fixpoint definition, thus permitting nondeterminism to be unbounded. The refinement relation is defined in a straightforward way.

As a test of our formalization, we have proved, among other things, Dijkstra's "healthiness conditions" (strictness, monotonicity and conjunctivity) of the wp function. Strictness and monotonicity were simple but conjunctivity was hard. In fact we could not prove conjunctivity of iteration in the case of unbounded nondeterminism (this seems to require the use of ordinal chains and transfinite induction).

We have also attempted to prove refinement laws, which various case studies have shown to be useful. Many general laws were easy to prove. Rules concerning loops (iteration) were more difficult. Refinement of assignment statements were also sometimes difficult to prove. This is because the semantics of assignments involves substitutions.

An important property of the refinement relation is subcomponent monotonicity: any subcomponent S of a program T can be replaced by another program fragment S', if $S <= S'$ holds. This method can be automated by considering the program T to be an application $(\lambda\ X\ .\ T(X))S$. We have written an ML function which takes two arguments: the theorem $\vdash S <= S'$ and the term $\lambda\ X\ .\ T(X)$ and which returns the theorem

$$\vdash\ T(S)\ <=\ T(S')$$

This shows that the principle of program derivation in small steps can be used.

So far we have only worked with very small examples. Besides aiming at larger examples we are also trying to make the theory more general. This involves extending the language with e.g., recursion and multiple assignments. The block construct has proved to be a problem; it seems necessary to index the refinement relation with a list of global variables in order to make the theory strong enough to handle local variables. This will also be a focus of future work.

2 Hardware Verification Session

Session Chair: John Herbert, Cambridge University

December 15, Morning

Speakers:

Luc Claesen	*CHEOPS: Interfacing HOL to CATHEDRAL*
Shiu-Kai Chin	*Combining Hardware Synthesis and Silicon Compilation*
David Sheperd	*SAFEMOS*
Phillip J. Windley	*Hierarchical Verification of Microprocessors*

CHEOPS: Interfacing HOL to CATHEDRAL

Luc Claesen
claesen@imec.be
IMEC, Belgium

A basic research action cooperation among IMEC, the HOL-group at the University of Cambridge and Philips (Dr. A. Kalker) will concentrate around the formal proof of steps in the CATHEDRAL-synthesis systems.

The CATHEDRAL synthesis systems are silicon compilers dedicated towards the automatic synthesis of digital signal processing systems. These systems are encountered in telecommunication, digital audio, radar etc.. Cathedral-1 is targeted towards bit-serial implementations of signal processing systems [1]. Cathedral-2 does the synthesis in multi-processor dedicated micro-programmed chip architectures [2]. The synthesis in Cathedral-2 starts from a high level algorithm specification in the applicative SILAGE language. In a first step, a customized datapath allocation is done with a tool called Jack-the-Mapper. This tools builds up the datapaths according to the requirements from the algorithm. It allows to make tradeoffs between area and speed. After the datapath allocation a scheduling of the operations on the datapath is done with the ATOMICS program. The number of busses can be reduced by bus merging at the cost of a few additional cycles. In Cathedral-2 a generic controller structure is used. After the scheduling the controllers are generated in the CGE program. At this point the interconnection of all physical building blocks is known and the layout phase can start. In Cathedral-2 a number of predefined parameterized modules exist (ALU, ACU, MULT, ...) that can be adapted in a very flexible way to the application at hand.

In this project the goal is to use the HOL system to try to verify the results of the synthesis processes in CATHEDRAL. The development of complicated proofs and theories will be payed off by their more frequent use in the cross check of a synthesis process. Even in more established so called "correct-by-construction" processes such as standard-cell or gate array layout generation, cross checks are done via tools such as netlist comparison. Probably not all of the tasks in the CATHEDRAL silicon compilation environment will be as tractable to be used for correctness proof by HOL. Therefore individual aspects of the CATHEDRAL system will be attacked first, starting with the parameterized module libraries. The future goal is to develop constructive methods, based on transformational design, that will result in proven synthesis steps in CATHEDRAL as well as in manual design steps.

References:

1. "Custom design of a VLSI PCM-FDM transmultiplexer from system specifications to circuit layout using a computer aided design system", R. Jain, F. Catthoor, J. Vanhoof,

B. Deloore, G. Goosens, N. Goncalves, L. Claesen, H. Van Ginderdeuren, J. Vandewalle, H. De Man, Joint special issue of the IEEE transactions on Circuits and Systems Vol. CAS-33, No.2, pp. 183-195, February 1986, and the IEEE Journal of Solid-State Circuits Volume SC-21, No.1, pp. 73-85, February 1986, on VLSI analog and digital signal processing.

2. "CATHEDRAL-II: A Silicon Compiler for Digital Signal Processing Multiprocessor VLSI Systems", H. De Man, J. Rabaey, P. Six, L. Claesen, IEEE Design and Test of Computers, Vol.3, Nr.6, December 1986, pp.13-26.

Combining Hardware Synthesis and Silicon Compilation

Shiu-Kai Chin
chin@sutcase.case.syr.edu
Syracuse University

One of the main objectives of this effort is to increase the level of machine-executable and machine-verified design knowledge. The rate of increase in the level of abstraction of design procedures must track corresponding increases in circuit integration to keep the level of human design effort for large systems manageable. The availability of machine-executable and machine verified design procedures enables designers to design at a higher level of abstraction with confidence. Towards this end, several ongoing efforts have been undertaken and are summarized below.

1. Verified design functions for inner product hardware.

   Several higher order functions used to create inner product hardware have been described and verified in HOL. The synthesis functions support general signed-binary representations and are parametric in wordsize and interconnection schemes. Some of the results are reported in [1,2,3].

2. Linkages to silicon compilers.

   One of our objectives is to use the above functions to create designs which are then layed out using a commercial cell library and silicon compiler. Currently, we are using Silicon Compiler System's Generator Development Tools (GDT). GDT supports both full custom, semi-custom, and standard cell design. GDT uses parametric cell generators to instantiate specific layouts and is capable of creating a specific layout from a register transfer level schematic consisting of standard and user defined cells.

   We also intend to incorporate the synthesis functions described above into Mike Fourman's LAMBDA system [4]. Our hope is that schematics generated by LAMBDA using verified design rules can be used as inputs for GDT.

3. An experiment verifying an abstract machine for a declarative language

As part of another project, we are developing a language which combines both functional and logic programming paradigms within a reduction setting. This language is an extension of J. Alan Robinson's LOGLISP language.

The abstract machine underlying the language [5,6] is a combination of the Three Instruct Machine (TIM) [7] and the Warren Abstract Machine (WAM) [8]. We have described the semantics of the language in [6] using Plotkin style operational semantics [9]. Proofs relating the abstract machine to the language have been done using the CLIO theorem prover.

Also, preliminary designs have been done to create hardware supporting the abstract machine. The VLSI designs have been done using the GDT tool set.

References

1. Chin, S.-K. and Stabler, E.P., "Synthesis of Arithmetic Hardware Using Hardware Meta-Functions", to appear in IEEE Transactions on Computer-Aided-Design, 8/90.

2. Chin, S.-K., "Verified Synthesis Functions for Negabinary Arithmetic Hardware", Applied Formal Methods for Correct VLSI Design, Luc Claesen Editor, Elsevier.

3. Chin, S.-K., "Combining Engineering Vigor with Mathematical Rigor," Hardware Specification, Verification and Synthesis: Mathematical Aspects, Editors: M. Leeser and G. Brown, Lecture Notes in Computer Science, Volume 408, Springer-Verlag, 1990.

4. Fourman, M. P., Harris, R. L., "Lambda - Logic and Mathematics Behind Design Automation," Abstract Hardware Limited, Uxbridge, UB8 3PH, U.K.

5. Jamsek, D., Greene, K.J., Chin S.-K., Humenn, P.R., "WINTER: Wams IN Tim Expression Reduction," Proceedings of the North American Logic Programming Conference, Cleveland, Ohio, October 16-19, 1989.

6. Jamsek, D.A., Chin, S.-K. Structured Operational Semantics for a Combined Function and Logic Programming Language, CASE Center Technical Report No. 8915, Syracuse University, Syracuse, NY, January 23, 1990.

7. Fairbairn, J., Wray, S., "TIM: a simple lazy abstract machine to execute supercombinators," Proceedings of the Functional Languages and Computer Architecture Conference, 1987.

8. Warren, D.H.D., "An abstract PROLOG instruction set," SRI International Technical Note 306, 1983.

9. Plotkin, G.D., "A Structured Approach to Operational Semantics," Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, September 1981.

SAFEMOS

David Sheperd
des@inmos.com
INMOS Limited, UK

Apology I started with an apology that I had expected to have several weeks in which to prepare material to present at the meeting but that this time had suddenly disappeared when I was given the task of sorting out the INMOS end of an ESPRIT project proposal. Hence my talk would consist of a brief introduction to the SAFEMOS project followed by some material I had prepared earlier but not used.

Project Reports

SAFEMOS The SAFEMOS[1] project is an UK DTI IED project whose partners are

- INMOS ltd
- SRI International Cambridge Research Center
- Oxford University Programming Research Group
- Cambridge University Computer Laboratory

The aim of the project is to "demonstrate the possibility of totally verified systems". This will be achieved by designing a language (similar to occam), a program verifier for that language, a verified compiler and a verified processor. Using these an verified program can be run securely using a verified processor and compiler. The project had just officially started and would last 3 years. Verification work would be based around HOL.

DECISIVE Also mentioned was the proposed ESPRIT DECISIVE[2] project. Proposed partners for this are

- Siemens
- SGS-Thomson Microelectronics

[1] SAFEMOS is *not* and acronym.
[2] DECISIVE is an acronym of DEsign of Correct Industrial Systems by Interactive VErification.

- INMOS Ltd
- Abstract Hardware Limited
- Harlequin Ltd
- Laboratory for the Foundations of Computer Science (Edinburgh University)
- CEAB

Part of this project would aim to take the results from projects like SAFEMOS and to produce industrially usable tools for verified design. Areas of interest at INMOS included verified synthesis of hardware and mixed hardware/software systems.

Design Methodology The main part of the presentation used some foils I had prepared some time earlier to describe how a consistent methodology could be used to design hardware in both a top-down and bottom-up manner which could be incorporated with a CAD design database.

The problem was that when designing a module that includes submodules you have to either already have designed the sub-modules, forcing design to be bottom up, or to use specifications of the submodules. If specifications are used then when implementations are produced a proof has to be performed to replace the specifications by implementations. Also, the designer needs to know which sub-modules are implemented and which are still just implementations.

The method suggested involved treating sub-modules as extra parameters to the module. Then the correctness statement of a module was of the form

> "For all sub-modules implementations
> (each sub-module implementation implies its specification)
> implies the module with those sub-modules implies its specification"

For example for a 4 bit adder with 1-bit adder submodules this would be in HOL

$$
\begin{array}{l}
\forall ADDER\_IMP \bullet \\
\quad (\forall x?, y?, cin?, s!, cout! : cout \bullet \\
\quad\quad ADDER\_IMP(x?, y?, cin?, s!, cout!) \\
\quad\quad \Rightarrow ADDER\_SPEC(x?, y?, cin?, s!, cout!)) \\
\quad \Rightarrow \\
\quad \forall cin?, cout : sig, x?, y?, s! : bus4 \bullet \\
\quad\quad ADDER4\_IMP\ ADDER\_IMP(x?, y?, cin?, z!, cout!) \\
\quad\quad \Rightarrow ADDER4\_SPEC(x?, y?, cin?, z!, cout!)
\end{array}
$$

The proof of this is much easier than might appear as suitable applications of rewriting, `STRIP_TAC` and `RES_TAC` reduce to the "normal" proof.

At some level the designer encounters true implemented modules. These have a correctness term of the form "implementation implies specification". Such implementations without sub-module parameters are termed "realizations" to indicate that they are for-real designs. A realization can be used in a higher level modules and by instantiating them in the parameter list of that module. Now using modus ponens with the specification term for the realization and that of the enclosing module one of the relevant sub-module assumption in the implementation can be removed. Once all sub-module parameters are replaced by realizations the module itself becomes a realization.

The benefits of this technique are

- "Realizations" are always genuine usable designs — a realization can only be constructed out of other realizations.
- Design can be top-down or bottom-up as sections can be left as specifications.
- Annotations were shown on how this method could be added to the INMOS HDL to allow for inclusion in CAD tools.

Semi-Custom Verification

Phillip J. Windley
windley@iris.ucdavis.edu
University of California, Davis

The terms "custom" and "semi–custom" have come to have particular meanings in the chip fabrication business. A "semi–custom" VLSI chip is one fabricated from pre–designed gate arrays, making it both cheaper to fabricate and less efficient (in space and time) than a fully custom design. We use these words in a analogous manner to describe the verification of a microprocessor that can be changed with little reverification cost, but with some lose of flexibility.

Formal verification promises to provide increased confidence that computer systems meet their specifications; but to realize this benefit, the cost and complexity of formal verification must be reduced. To date, verification has primarily been used in safety–critical and security–critical computer systems where the cost and complexity can be justified. Other systems, however, could benefit from the application of formal methods if simpler means of verifying hardware and software can be found.

Several microprocessors have been verified using formal methods. Each of these verifications has been "full–custom," that is, each microprocessor design and subsequent verification was done to some custom specification. Because of the complexity of verifying real computer systems, the only successful verification efforts have been on relatively simple microprocessor architectures (fewer than 32 words of micro–instruction store, small instruction set, limited features for supporting operating systems, etc.). The largest of these microprocessors, the VIPER microprocessor, was abandoned due to the large amount of human effort seemingly required to complete the proof.

This talk discusses the formal verification of a microprocessor, but more importantly, we present a theory of interpreter hierarchies that provides a means of significantly reducing the complexity of microprocessor verification. This theory is embodied in a set of tools and techniques for the HOL theorem prover. We present the theory and a discussion of how this theory leads to the "semi–custom" verification of microprocessors making formal methods more accessible to the design engineer. For a more detailed treatment of this research see [1,2].

Reference:

1. Windley, Phillip J.,"A Hierarchical Methodology for Verifying Microprogrammed Microprocessors," University of California, Davis Division of Computer Science Research Report No. CSE-89-27, November, 1989.

2. Windley, Phillip J, "The Semi-Custom Verification of Microprocessors," University of California, Davis Division of Computer Science Research Report No. CSE-90-1, January, 1990.

3 Foundations Session.

Session Chair: Tom Melham, Cambridge University

December 15, Afternoon

Speakers:

Ton Kalker	*Well-ordering and Cardinals in HOL*
Elsa Gunter	*Abstract Theories for HOL*
Sten Agerholm Henrik Skjoedt	*Automating a Model Checker*
Stig Pedersen	*Modeling of Diagnostic Reasoning*

Well-ordering and Cardinals in HOL

Ton Kalker
kalker@apolloway.prl.philips.nl
Philips Research Laboratories, The Neatherlands

The notion of a well-ordering as defined in ordinary set theory is recalled: A linear ordering on a set S is called a well-ordering if every non-empty subset E of S has a smallest element.

A famous theorem of Zermelo (1904) asserts the fact that the axiom of choice is equivalent to every set having at least one well-ordering. As in the HOL logic the axiom of choice is automatic by the presence of the Hilbert choice operator, the above mentioned set-theoretical fact can be translated to the HOL logic. We showed how set theory can be "simulated" in HOL by representing sets as predicates on types, and then gave an outline of the prove. The complete prove can be found in the library `well_ordering` of HOL88.

Having `well_orderings` available gives us the tool for formulating the transfinite-induction principle, which is a straightforward generalization of the general induction tactic for naturals. We showed the formulation of this induction tactic and mentioned the associated tactics and rules, which can also be found in the library `well_ordering`.

As an application of the library `well_ordering` and the library `quotient` we did a short outline of a theory about cardinals. We showed how with the help of a quotient construct cardinals can be defined: they are just classes of equipotent sets, where as before sets are modeled as predicates on types. Set inclusion is extended to a partial order on cardinals. To prove the fact this this ordering is even linear, we need transfinite induction. Transfinite induction is used to show that:

> if set A can not be surjected onto set B,
> then
> there exists an embedding of A into B.

We closed the talk with saying that the full details can be found in the library of HOL88, and that the work done showed that HOL can be used for non trivial math.

A paper on well-orderings, cardinals, etc. in HOL will appear shortly.

Abstract Theories for HOL

Elsa Gunter
elsa@linc.cis.upenn.edu
Univeristy of Pennsylvania, USA

In this talk an extension to the HOL theory mechanism is proposed, namely that of abstract theories. An abstract theory extends the current notion of theory in HOL by allowing the user to abstract out of a theory type variables, term variables and theorem assumptions. Within a theory so bound, the bound type variables have the status (for almost all purposes) of 0-ary type constructors, the bound term variables have the status (again, for almost all purposes) of monomorphic constants, and the bound theorem assumptions have the status of axioms. From outside the abstract theory these are bound type variables, bound term variables and bound theorem assumptions, all awaiting instantiation. The goal of adding such a construct to HOL would be to facilitate the natural development and use within HOL of abstract mathematical theories with repeated application.

The attributes of an abstract theory are to be:

- Type Binder (list of type variables)
- Term Binder (list of term variables (with their types))
- Theory Assumptions (list of pairs of strings and terms of type bool)
- Parent Theories (list of theories and abstract theories)
- Type Constructors
- Constants
- Axioms
- Definitions
- Theorems

The functions for interaction with abstract theories are to be as follows:

- For declaring a new abstract theory:

```
new_abstract_theory: string -> type list -> term list ->
                     (string # term) list -> void
```

where

- `string` is the name of the abstract theory.
- `type list` is a list of type variables to be treated (almost) as type constants from within the abstract theory.
- `term list` is a list of term variables to be treated (almost) as monomorphic constants from within the abstract theory. The types of the term variables can only involve those variables found in the type list.
- `(string # term)` list is a list of labeled propositions which are treated as axioms from within the theory. The propositions may involve those type variables and those free term variables appearing respectively in either the type list or the term list.

- For extending or loading an abstract theory:

  ```
  extend_abstract_theory: string -> void
  ```

  ```
  load_abstract_theory: string -> void
  ```

  These commands allow for the addition of definitions, type definitions, and theorems to previously created abstract theories.

- For closing an abstract theory:

  ```
  close_abstract_theory: void -> void
  ```

  This command returns HOL to previous concrete theory, and removes all ML identifier bindings made while the abstract theory was open.

- For instantiating a theorem:

  ```
  extract_theorem: string -> string -> thm.
  ```

  where

  - The first `string` is name of abstract theory.
  - The second `string` is name of theorem in abstract theory.
  - The `thm` returned is the result of discharging all the abstract theory assumptions and generalizing all the the variables in the the term binder for the theorem named.

- For instantiating all of the abstract theories:

```
apply_abstract_theory: string -> string -> type list -> term list ->
                        thm list -> void
```

  This command incorporates into the current theory the results of instantiating all of the theorems in the abstract theory with the values given in type list and term list, provided that

  – The first `string` is the name of the abstract theory.
  – The second `string` is a modifier to the names of the theorems from the abstract theory.
  – The `type list` provides values to instantiate the type variables in the type binder of the abstract theory.
  – The `term list` provides values of the appropriate instantiated type to instantiate the term variables in the term binder.
  – The `thm list` provides theorems whose statements are the instantiated versions of the abstract theory assumptions.

The following functions are used from within an abstract theory:

- The function

```
thy_asm: string -> thm.
```

  returns as a theorem one of the theory assumptions, provided that `string` is a label for the theory assumption.

- For introducing new type operator:

```
abs_new_type_definition: (string # term) -> thm
```

  where

  – `string` is the name of the new type operator to be introduced.
  – `term` is a predicate (containing no free variables) on an existing type which defines a subset which, if non-empty, is to be in one-one correspondence with the type being defined,

  This function introduces a new type operator with arity the same as the arity of the type on which P is a predicate (for this purpose the types and terms in the binder of the theory are to be considered as free), ana which asserts an axiom stating

```
|- ?rep.(!y z. (rep y = rep z) = (y = z)) /\
        (!x. (P x \/ (x = @y. P y)) = (?z. x = rep z))
```

which is returned as a theorem.

- In addition, we would have the usual functions for internal interaction with concrete theories as currently existent, except that `new_type_definition` is prohibited, and new_definition is required to not define a constant whose name is that of a term in the theory binder. Within `new_definition`, the terms of the theory binder are disallowed.

In conclusion, the proposed notion of abstract theories provides a modest, consistent extension to HOL, which we believe will provide a convenient environment for developing and applying theories with assumption. In conjunction with the existing theory capacity of HOL, they should play the same role for theorem proving in-the-large as structures and functors (i.e. "modules") due in Standard ML. Moreover, they should remove *all* excuses for non-foundational uses of new-axiom.

Automating a Model Checker for Recursive Modal Assertions in HOL

Sten Agerholm
Henrik Skjoedt
dathsk@daimi.dk
Aarhus University, Denmark

Transition systems are widely used in computer science, communication engineering and circuit designing, among others, for modeling the behavior of both hardware and software systems. For example, computer scientists use transition systems in studying finite state concurrent programs; the operational semantics of such can be given by finite labeled transition systems.

Properties of states in finite labeled transition systems can be expressed using recursive modal assertions. The assertion language we have implemented in the HOL system includes recursive assertions defined as maximum and minimum fixed points. This language is powerful and can be used to express properties such as 'always', 'eventually' and 'deadlock', among others. In order to have the recursive assertions in HOL we proved a special version of Tarski's theorem which provides a way of defining and reasoning about maximum and minimum fixed points.

In general it can be quite hard work to prove by hand whether or not a state satisfies an assertion, and it is therefore desirable to have a way of checking whether assertions are true or false on states mechanically. We have implemented an algorithm which proves theorems in HOL, automatically, stating whether a state satisfies an assertion or not.

The model checking system consists of various tools which reduce the workload of the user to a minimum, for example the system does all the theorem proving needed. In addition, it is made flexible with respect to efficiency optimizations.

Modeling of Causal and Diagnostic Reasoning

Stig Andur Pedersen
sap@meza.ruc.dk
Roskilde University Center, Denmark

Clinical reasoning is rendered difficult by the logical complexity of causal explanations of various diseases. A logical analysis of causal structures involved in clinical reasoning is presented. Disease causes are chosen as non-redundant parts of causal complexes. A causal complex is a description of a situation which by necessity leads to the disease. Usually causal complexes are identified by listing relatively few non- redundant components of it. It is argued that counterfactual conditionals (i.e., conditionals with false antecedents) play a decisive role during the construction and analysis of causal complexes and that we would gain a better understanding of causal reasoning by developing formal analyses of counterfactuals involved in such reasoning. One such formal analysis is sketched. We believe that HOL will be a suitable tool for implementing and further analyzing this logical system.