Third Annual HOL User Meeting
1 - 3 October 1990

Day 1, Monday, 1 October 1990
Morning Session

10:15 - 10:45
Recent Developments in the HOL world.

Mike Gordon
Cabridge University, UK

- HOL88.1.12 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
- π–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

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

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

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 Melham 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 π–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 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 be written

- **Status:**
  Funded by SERC

- **Researcher:**
  Avra Cohn

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

SAFEMOS I

- **Research Goal:**
  - Demonstrate the possiblilty of totally verified systems by creating a verified application program running on a verified processor by means of a verified compiler
  - Develop a methodolgy 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

10.45-11.15
A Short Overview of Version 1.12

Tom Melham
Cambridge University, UK

The most recent version of the Cambridge HOL system is version 1.12, which will be ready for general release sometime early in January 1991. This talk consisted of an overview of some recent changes made to the system for version 1.12. The main points are summarized below.

Preterms

An ML abstract type `preterm`, values of which represent parse trees of HOL terms, has been added to the system. This experimental feature is intended to provide the input type to user-programmed HOL type checkers. This is done by providing a flag `preterm` and an assignable ML variable `preterm_handler` of type `preterm->term`. When the flag `preterm` is set, quotations evaluate to preterms, which are then filtered through the user-supplied function bound to `preterm_handler`. A built-in function `preterm_to_term` is provided that implements the standard type checking algorithm.

Revised resolution tools

The resolution tactics: `RES_THEN`, `IMP_RES_THEN`, `RES_TAC`, and `IMP_RES_TAC` are modified in version 1.12 to include:

`!x. t1==>t2` $\Rightarrow$ `t1==>!x.t2` (x not free in `t1`)

in the transformations that are done (by `RES_CANON`) when a theorem is being prepared for resolution. This prevents specialization of variables from happening too early during the process of adding assumptions.

Unfortunately, this change is *not* compatible with previous versions of HOL. Existing goal-directed proofs may not run in the new version, particularly if they are sensitive to the exact ordering or number of assumptions in the goals. But it was thought that the benefits of handling universal quantitifiers properly would outweigh the irritation of some users having to re-run their proofs. Resolution is likely to change even more before version 1.12 is 'officially' released.

Paired beta-conversion

A new conversion called `PAIRED_BETA_CONV` has been added to do generalized beta conversion of tupled lambda abstractions applied to tuples (i.e. tupled redexes). The conversion works for arbitrarily nested tuples.

Natural number division

The built-in theory arithmetic has been augmented with a minimal yet useful theory of the functions `MOD` and `DIV` on the natural numbers. A number of additional arithmetic theorems (mostly about subtraction) have also been added.

Type definition package

The code for built-in function `define_type` has been totally rewritten. The main visible change is that this function now has a proper parser for its type specification argument. In particular, the syntax of types accepted by `define_type` is now identical to that of logical types in quotations. Most of the associated functions for reasoning about recursive types (e.g. `INDUCT_THEN`) have also been completely rewritten.

Revised set theory library

The `sets` library (formerly called '`all_sets`') has been extensively revised and extended. The library has been restructured, and several new built-in theorems are available. In addition, parser and pretty-printer support for notation like:

`{x1,...,xn}`	=	a finite set
`{x | P[x]}`	=	the set of all `x` such that `P[x]`
`{t[x] | P[x]}`	=	the set of all `t[x]` such that `P[x]`

is now provided as a built-in feature. Proof support for this notation is supplied in the form of a conversion that implements the axiom of specification for sets specified using it. The library also makes available an induction principle (i.e. a tactic) for proving properties of *finite* sets.

The contrib directory

As was discussed at the second HOL user's meeting, a '`contrib`' directory has been instituted for the HOL distribution tape. This directory contains contributions from the HOL user community which are distributed with the HOL sources. The contributions are not edited, and the University of Cambridge Computer Laboratory assumes no responsibility to support any HOL code or tools distributed in this directory.

The aim of `contrib` is to provide a vehicle to make it easy for the HOL community to share theories, proofs, examples, tools, documents, and other material which may be of interest to users of the HOL system, but which is not included in the library.

Each contribution is given a separate subdirectory within `contrib`. A standard `READ-ME` file is included in each subdirectory, giving at least the following information:

- name of the subdirectory
- a one-line description of the contents
- names and addresses of the authors
- date of inclusion in `contrib`

Contributors may also include other information in the `READ-ME` file. Contributed material from users is most welcome.

Hiding/deleting of functions

A large number of built-in ML functions which were available at top-level in previous versions of HOL and which were never intended for general use have now been either deleted or hidden from the user. These include functions or other ML values which were redundant (e.g. `AND_CLAUSE1`), obsolete (e.g. `OLD_RES_TAC`) or unused (e.g. `form_vars`).

Present status

Version 1.12 contains a great many small changes in addition to those mentioned above, all of which are documented in a file included on the distribution tape. Version 1.12 is almost ready for general release, and should be available from early January 1991.

11:30-12:00
A Re-implementation of hol88 in Standard ML

Konrad Slind
University of Calgary

1. **hol88 and reasons for implementing it in Standard ML.**
   The main reasons are better chances at maintenance and extension, and to take advantage of the good SML compilers that are becoming available.

2. **hol90 status.**
   It is mostly done, but there are a lot of little things that need to be solved. It provides the same user interface modulo a swap in the quotation and string delimiters, and the embedding in Standard ML.

3. **Standard ML issues, or why I used PolyML.**
   PolyML provides a facility for installing a new top level, for installing pretty printers for terms, theorems, and goals, and NJSML doesn't. We plan to add in structures before releasing the system.

4. **A tour of the implementation.**
   How hol90 builds, i.e., the dependencies when the system is constructed.

5. **Interesting facts that I dug up in the process of implementing the system.**

6. **Speed revisited**
   Some favorable statistics on the speed of the system are given.

7. **Conclusion.**
   There is a lot of work to be done yet before we have a replacement for hol88

8. **ML to SML conversion guide.**
   Some rules we found useful when doing the translation from ML to SML.

12.00-12.30
A High-Assurance Implementation of HOL.
R.D. Artman
ICL, UK

SHORT ABSTRACT:
ICL are engaged inan IED sponsored project to develop a highly assured version of HOL for use in industrial applications.

Day 1, Afternoon Session

14.00-15.00
The Implementation and Use of Abstract Theories in HOL

Elsa L. Gunter
AT&T Bell Laboratories

What do we want from abstract theories?

An abstract theory should create a context of assumed information, and give that information scope. Within the abstract theory, attempts to prove theorems have access to that information. Outside the 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 it, one must supply an instance of the abstracted information.

More specifically, an abstract theory should take a signature consisting of

- a list of type variables
- a list of term variables
- a list of theory 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 the assumptions are treated as axioms. Outside the abstract theory, the type and term variables should behave only as variables, and the assumptions should no longer be treated as axioms. To access a theorem of an abstract theory we must supply instantiations for type and term variables, and proofs for the assumptions.

Abstract theories will provide us with the ability to do much of abstract mathematics in a clean and straight-forward manner. Another advantage of abstract theories is that it allows for modularity in the development of a complex theory. It could, for example, be used in organizing verification proofs to make the implementation-specification layers clearer.

How do we implement this?

First attempt:

One way of "implementing" abstract theories is to add the theory assumptions to each goal to be proved, and do nothing with the type variables and term variables except expect them to be used in a consistent manner. This is essentially what was done with Group Theory.

So what is wrong with this approach? Consider the abstract theory of monoids. Let `Is_a_Monoid(M,prod,id)` be a predicate which is equal to the conjunction of the monoid axioms for `M`, `prod` and `id`. From the axioms we can prove

```
Thm1 = [M m, Is_a_Monoid(M,prod,id)] |- prod id m = m
```

Suppose we have

```
Thm2 =
 [M (prod x y), ..., Is_a_Monoid(M,prod,id)] |- prod id (prod x y) = w
```

which we want to rewrite with `Thm1`. The result should be

```
Thm2' =
 [M (prod x y), ..., Is_a_Monoid(M,prod,id)] |- prod x y = w
```

with substitution theorem

```
Thm1' =
 [M(prod x y), Is_a_Monoid(M,prod,id)] |- prod id (prod x y) = (prod x y)
```

But it should not be possible to further rewrite to

```
Thm2'' =
 [M (prod x y), ..., Is_a_Monoid(M,prod,x)] |- y = w
```

with substitution theorem

```
Thm1'' =
 [M y, Is_a_Monoid(M,prod,x)] |- prod x y = y
```

We should not be able to instantiate `id` with `x`.

Within an abstract theory, we should not be able to

- instantiate the type variables in the theory signature,
- instantiate the term variables in the theory signature,
- discharge the theory assumptions from a theorem.

Outside an abstract theory we should be able to do all these.

Second attempt:

Another way of implementing abstract theories, is to convert the type variables and term variables of the signature into constants within an abstract theory, and then when saving a theorem translate them, to change these special type and term constants back into type and term variables, and to append the theory assumptions to the theorems' assumption list.

With this approach, definitions become a nuisance. We must restrict our definitions to not using any of the special type or term constants. For example, if in the case of monoids discussed above we wish to define the squaring function, we cannot make the definition

```
Square_DEF = |- Square x = prod x x
```

but instead must say something like

```
Square_DEF = |- Square prod' x = prod x x
```

This requires rewriting the functions `new_specification` and `new_definition` and recompiling all the functions that depend on them.

Another difficulty that we face concerns the scope of entities created in a abstract theory. Inside an abstract theory we have access to special type constants, term constants and axioms which are not valid entities outside the scope of the abstract theory. Theorems built using them are only valid in the presence of the assumed axioms; outside that scope such theorems are not valid and can lead to inconsistencies. However, particularly in the presence of references, anything of type `thm` can live as long as the HOL session does. One way to deal with this problem is to force the HOL session to end upon the exiting of an abstract theory.

This approach has been implemented in HOL.1.11 as a separately loadable library. It involves only a modest modification of the underlying system and redefinition of ML-HOLfunctions. One caution that any user of the library must observe is that since the functions `new_specification` and `new_definition` must be redefined, any functions that are derived from these that the user has added will need to be recompiled.

Third attempt:

The last approach to implementing abstract theories is the most radical of all. It involves creating within HOL a formal notion of a context which consists of a list of type variable and term variable which are to be consider as bound in the context, and a list of terms of type `bool` which the context will allow to be treated as axioms. We will then change basic datatypes for types, terms and theorems in HOL so that they all contain a context that they need to be interpreted relative to. The various functions for constructing types, terms and theorems will need to be modified to be sensitive to these contexts. For example, `INST_TYPE` will need to be modified so that type variables bound in the context of a theorem cannot be instantiated, and `ABS` will need to be modified so as not to abstract term variables that are bound in a theorem's context. This will give a consistent extension to the current HOL system and all the old proofs in the regular theories will still be valid. Types, terms and theorems as they currently exist will become types, terms and theorems having empty contexts (*i.e.* the lists of type variables, term variables and "axioms" are all empty).

This approach has the drawback that it requires much fundamental rewriting of the HOL system. It is definitely not going to be an add-on library. I anticipate carrying out this approach in the Standard ML implementation of HOL that is being produced at Calgary.

15.00-15.30

A Tree-editor Interface to HOL

Myla M. Archer, joint work with George Fink

Division of Computer Science

University of California, Davis

PM is a tree-editor-based proof manager that provides an interface to HOL. It is undergoing development at the University of California, Davis. A beta version will soon be made available (contact `gfink@eecs.ucdavis.edu` or watch for an announcement in `info-hol`).

The underlying tree editor is Tree-mode, a programmable tree editor package for Gnu Emacs developed by Samuel Kamin and David Hammerslag at the University of Illinois. The programmable features of Tree-mode allow one to customize such things as subtree display, side-effects of editing operations, and interfaces to outside programs (such as HOL).

As an HOL interface, PM builds and maintains trees that correspond to proofs. It maintains a theory node as the root of the tree; each child of this theory node holds a potential theorem, and is the root of the proof tree of this theorem. All the non-theory nodes – the children of the root theory node, and the descendants of these children – contain an assertion and a *method* (*i.e.*, tactic). They also contain an indication of whether the node is `validated`.

Currently, tree expansion from a non-theory node is done automatically by PM; provided the method is valid, a call to HOL will result in adding appropriate subgoals to the node, and marking the node as *validated*. Tree contraction currently can be done under two circumstances. If there is a purely linear part in a proof tree, this part can be *squashed* into one node, whose assertion will be that of the immediate parent of the linear part, and whose method will be determined from its original method and those of its eliminated descendants. A completely validated tree can similarly be squashed into a single node, where again, the assertion is that of the root, and the method is found from the methods at all the nodes in the subtree.

PM provides a number of conveniences to HOL users absent in the original interface. For example, subgoals anywhere at the frontier of the proof tree are accessible to a user (and, in fact, proofs at arbitrary internal nodes can be revised at the user's convenience). Tactics used in the proof are automatically kept track of, and a complete proof can be automatically generated from a completely validated proof tree. Lemmas used in a proof can be easily extracted and added to the underlying theory.

We are planning some additional features for added convenience in expanding and contracting proof trees. One feature would allow contraction of any completely validated tree segment into a single node, with the nodes at the frontier of this tree segment as the new children. This would increase one's ability to save a proof tree in a form showing only the major decision points. A second feature would allow one to add children to a node by hand, and then use an appropriate `method` to validate the node, normally resulting in additional subgoals as children. This provides another way to reuse lemmas.

An essential feature of PM that we expect to maintain is its soundness as an extension to HOL. Care is taken to be sure that editing operations do not result in nodes being marked `validated` when they should not be. Thus, any completely validated proof tree constructed using PM will correspond to a valid HOL proof.

16.00-16.30
A Mutually Recursive Types Package in HOL.

Poul Christiansen, Hans Jakob Pedersen & Anders Pilegaard, Peter Strarup Jensen, Ann-Grete Tan
Aarhus University, Denmark

New logical types in HOL are defined in terms of existing types. The procedure is straightforward using the ML functions `new_type_definition` and `EXPAND_TY_DEF`. It involves defining a *representing* subset of an existing type using a predicate, and then proving that the predicate is true in at least one case. This is to adhere to the HOL restriction that types must be non-empty, because of the Hilbert ϵ-operator. `new_type_definition` and `EXPAND_TY_DEF` take this information and complete the definition process, finally returning a theorem that *axiomatises* the new type.

Tom Melham's `define_type` function has simplified this procedure for (non-mutually) recursive and simple enumerated types. When given a (single production) grammar specifying the elements of such a type, `define_type` returns a primitive recursion theorem axiomatising the type. For example:

```
#let tree =
     define_type `tree` `tree = Leaf num | Tree tree tree`;;

tree =
   |- !f0 f1.  ?!  fn:tree->*.
     (!x.  fn(Leaf x) = f0 x) ∧
     (!t t'.  fn(Tree t t') = f1 (fn t) (fn t') t t')
```

A call of `define_type` results in the instantiation of a pre-proved general theorem.

We have generalised this functionality by defining `define_mutually_recursive_types` which accepts a multiple production grammar, specifying an arbitrary number of mutually recursive types, eventually returning a single axiomatisation of all the types. For example:

```
#let UBtree = define_mutually_recursive_types
   (`Exp`,
     [`Utree = ULeaf num | UTree Btree`;
      `BTree = BLeaf num | BTree Utree Utree`]);;

UBtree =
   |- !f0 f1 f2 f3.
     ?!(fn0:Utree->*) (fn1:Btree->**).
       (!n.  fn0(ULeaf n) = f0 n) ∧
       (!E. fn0(UTree E) = f1 (fn1 E) E) ∧
       (!n.  fn1(BLeaf n) = f2 n) ∧
       (!E E'. fn1 (BTree E E') = f3 (fn1 E) (fn1 E') E E')
```

Its implementation relies heavily on both the standard type definition facilities, and on `define_type`. At each call the following steps are carried out automatically:

1. A new single production grammar is derived from the specification, by renaming all constructors, and concatenating the right hand sides. This specifies the representing type for all the mutually recursive types. `define_type` is used to define it.

2. For each mutually recursive type, a subset predicate is generated and proved to be true for a simple case. `new_type_definition` and `EXPAND_TY_DEF` are then used to define them individually.

3. A primitive recursion theorem for all the mutually recursive types is derived to axiomatise them. This theorem is useful for defining recursive functions on the new types, and for deriving an induction theorem. It is this theorem which is returned.

   A function `new_mutually_recursive_definition` has been defined to facilitate recursive function definitions.

This facility has been implemented by two groups of students at Aarhus University, Denmark (independently of each other!):

- Poul Christiansen (`pchris@daimi.aau.dk`) &
  Hans Jakob Pedersen (`sjakob@daimi.aau.dk`)

- Anders Pilegaard (`andersp@daimi.aau.dk`)
  Peter Strarup Jensen (`strarup@daimi.aau.dk`)&
  Ann-Grete Tan (`agt@daimi.aau.dk`)

16.30-17.00
Another Iteration in Arithmetic
Malcolm Newey

The Australian National University, Australia

The problems I have felt with natural number arithmetic, as it appeared in HOL88.1.11, were as follows:

1. In my applications, many simple theorems that I needed were not supplied; in many cases a lot of time was needed to prove them. (There were also theorems taking up name-space that were redundant or should have been in other theories.)

2. Being a slow user who is not able to remember many names I found that three theories to search through was too many.

3. The current theories claim the constants `0`, `1`, `+`, `-`, `*`, `<`, `>`, `<=`, `>=` *etc.* for use with natural numbers.

4. I do not have an interest in theorems that mention SUC but the current theory is biased in that direction.

I have worked on an alternative treatment of natural numbers which rests on the same axioms as in release 1.11 but which incorporates the following remedies:-

1. To have just 2 theories - `:nat` and `:nat-arith`. `:nat` has constants for `0`, `1`, `<`, `>`, `<=`, `>=` and *prim_rec*. `:nat-arith` has constants for `+`, `-`, `*`, `div`, `mod`, `exp`.

2. To give the actual constants distinctive names - `nat_0`, `nat_add`, *etc.* (`0`, `1`, `+` *etc.* are therefore not pre-empted). Use interface maps to print `nat_0` as 0, `nat_add` as + *etc.*

3. To develop a much more substantial theory with the following properties

   (a) It is sensibly organised,

   (b) All theorems have NAT prefix,

   (c) No SUC theorems,

   (d) Few theorems are "trivially derivable" from any other.

As well as increasing the richness of the theory, I have taken some steps in the direction of automating the theory by writing tactics:-

1. Many theorems of `:nat` can be used as basic rewrites; (NAT-REWRITE-TAC *ThL*) uses a standard set of such together with theorem list *ThL*.

2. (NAT-INDUCT-TAC *ThL*) is more general than "SUC induction", does not use SUC, and uses (NAT-REWRITE-TAC *ThL*).

3. (NAT-POS-TAC t *ThL*) does cases on $(t = 0)$, expands the assumption list of the "false subgoal" with $(\sim (0 = t))\&(0 < t)$, and does (NAT-REWRITE-TAC *ThL*).

4. (NAT-COMPARE-TAC $(t1, t2)$ ThL) gives subgoals corresponding to $(t1 < t2)$, $(t1 = t2)$ and $(t2 < t1)$; In each case, the assumption list is expanded by appropriate useful consequences of this last assumption; (NAT-REWRITE-TAC ThL) is then used.

5. (NAT-TRANS-TAC $(t1, t2)$ ThL) expects to expand the assumption list by making deductions about the relationship between $t1$ and $t2$.

I reported on the benefits and penalties (in space & time) arising from using these tactics. Some discussion of the prospects for this iteration of natural numbers followed.

Day 2, Tuesday, 2 October 1990
Morning Session

9.30-10.30
A Mechanized Theory of the Pi-calculus in HOL

Tom Melham,
Cambridge University, UK

The π-calculus is a process algebra in the style of CCS, developed by Milner, Parrow and Walker [1] for modelling concurrent systems in which the pattern of interconnection between processes can change over time. This talk described work in progress on a mechanized formal theory of the π-calculus in higher order logic using the HOL theorem prover. The main aim of this work, which is being done jointly with Mike Gordon at the Computer Laboratory in Camdridge, is to provide a tool to support both reasoning about applications using the π-calculus and metatheoretic reasoning about the calulus itself.

Four general principles are adopted in this project. First, a purely *definitional* approach (i.e. no non-definitional axioms) is being used. Second, we wish to automate wherever possible. In practice, this has so far meant providing efficient derived inference rules for proving simple syntactic propositions (e.g. about alpha-equivalence of terms in the π-calculus). The third principle is to make the HOL proofs as 'robust' as possible, in the sense that they should run even when minor changes are made to the definitions of basic concepts of the π-calculus. By this means, we hope to facilitate experimental investigations of variants of the calculus. Finally, we are trying to mechanize the π-calculus exactly as it is presented in [1]. That is, we wish to avoid simplifying the calculus merely in order to make the job of mechanizing it easier. In the talk, at least one point at which we have compromised this last principle was discussed.

The talk began with a very brief overview of the π-calculus, in which familiarity with process algebras (in particular CCS) was assumed. A formalization in HOL of the syntax of the calculus, which was done in the obvious way using the recursive types package, was then presented. It was then explained how the labelled transition relation for the π-calculus can be expressed in logic using a definitional higher-order inductive definition. Allusion was made to a forthcoming HOL tool for automating such definitions. (At the time of writing—Dec 1990—this tool is within a week or so of being finished to 'production' standard, and is now being used in the π-calculus project). The derivation from this definition of proof rules and a principle of 'rule induction' for the labelled transition relation was briefly discussed. A higher-order definition in HOL of the bisimulation relation for the π-calculus was then presented, from which it is proposed to derive an equational theory of equivalence in HOL. The talk concluded with a summary of progress to date.

References

[1] Milner, Parrow, and Walker, 'A Calculus of Mobile Processes', Laboratory for Foundations of Computer Science, University of Edinburgh, LFCS Report Number ECS-LFCS-89-85/86.

11.00-11.30
A Definitional Theory of UNITY in HOL
Flemming Andersen
TFL (A Telecomms Research Laboratory)

A Definitional Theory of UNITY in HOL

Higher Order Logic is powerful enough to model the inductive definition of the UNITY logic as defined in [CM88].

It is shown that the UNITY relation **leadsto** can be defined by specialising a general progress relation R_{close}, which models the transitive and disjunctive closure of binary relations R_b describing the state transitions of a program.

Introduction

UNITY is a theory for specifying and verifying properties about parallel programs. The theory consists of a simple programming language for specifying programs and a logic to express safety and progress properties which a program must satisfy. A program in UNITY consists of an initialisation section and a set of conditional statements. The UNITY logic in principle only defines one safety property called **unless** and two progress properties named **ensures** and **leadsto**.

We concentrate on the **leadsto** progress property. It turns out that the **leadsto** property does not need to be defined inductively as done in [CM88]. It can be defined as a pure definitional theory within Higher Order Logic.

All proofs of theorems presented in this description have been proved in the HOL system but are intentionally left out in this summary.

Progress in State Transition Systems

We may understand a program Pr as a state transition system in which every statement $st \in Pr$ is a transition from a state s to a new state $(st\, s)$.

To describe progress properties of a program we may define a basic progress relation R_b in p and q which prescribes that for any state s satisfying the predicate p a number of state transitions eventually leads to a new state s' satisfying q.

However, we would like to define an extended progress relation which takes sequencing and state space splitting into account.

This means, we must define a relation R_{close} which satisfies that:

$R_b\ p\ r_1$, $R_b\ p\ r_2$, ..., $R_b\ r_n\ q$, yields $R_{close}\ p\ q$,

and

$R_b\ p_1\ q$, $R_b\ p_2\ q$, ..., $R_b\ p_n\ q$), yields $R_{close}\ (p_1 \vee\ p_2\ \vee\ ... \vee\ p_n)\ q$.

If we want to define the relation R_{close} as the least relation that satisfies the three properties as described above, an inductive definition of the properties would be:

- Basic Closure
  $R_b\ p\ q \vdash R_{close}\ p\ q$
- Transitive Closure
  $R_{close}\ p\ r,\ R_{close}\ r\ q \vdash R_{close}\ p\ q$
- Disjunctive Closure
  $(\forall\, i : R_{close}\ P_i\ q) \vdash R_{close}\ (\exists_i :\ P_i)\ q$

Defining the Progress Relation R_{close}

To define R_{close} as described we begin by defining the family of relations $RelFam$ as the family of relations R_{close} that satisfies the three above mentioned properties:

$$\begin{array}{l} RelFam\ R_b\ R_{close}\ = \\ \quad \forall\, p, q : \\ \quad\quad (R_b\ p\ q\ \Rightarrow\ R_{close}\ p\ q)\ \wedge \\ \quad\quad (\forall\, r :\ R_{close}\ p\ r\ \wedge\ R_{close}\ r\ q\ \Rightarrow\ R_{close}\ p\ q)\ \wedge \\ \quad\quad (\forall\, P :\ (\forall\, i : R_{close}\ P_i\ q)\ \Rightarrow\ R_{close}\ (\exists_i :\ P_i)\ q) \end{array}$$

Now we may define the relation R_{close} of R_b in p and q as the intersection of all the relations R in $RelFam$:

$$R_{close}\ R_b\ p\ q\ = (\forall\, R :\ RelFam\ R_b\ R\ \Rightarrow\ R\ p\ q)$$

We want to prove that R_{close} is the least relation in $RelFam$.

First we prove that R_{close} is really a member of $RelFam$:

$$\vdash\ \forall\, R_b :\ RelFam\ R_b\ (R_{close}\ R_b)$$

Then we prove that R_{close} is less than any relation R which is a member of $RelFam$:

$$\vdash\ \forall\, R,\ R_b :\ (\forall\, p,\ q :\ R_{close}\ R_b\ p\ q\ \Rightarrow\ (RelFam\ R_b\ R\ \Rightarrow\ R\ p\ q))$$

Hence R_{close} is the least relation which is a member of $RelFam$.

We may easily prove that the progress relation R_{close} satisfies the three given properties.

Defining the leadsto Relation

In the UNITY theory the relation **ensures** is a basic progress property, and **leadsto** reflects the extended progress property. Hence, we may define **leadsto** by specialising R_{close} with **ensures**.

$$(p\ \mathbf{leadsto}\ q)\ Pr\ =\ (R_{close}\ (\lambda\, p\, q.\ (p\ \mathbf{ensures}\ q)\ Pr)\ p\ q)$$

Conclusion

The presented method of defining program properties as a definitional theory in higher order logic has given much confidence in the soundness of the UNITY logic. In addition it has allowed for a mechanization of the UNITY theory using the HOL system.

In [CM88] two different induction principles for the **leadsto** definition is used but not formalised. One of the used induction principles in [CM88] follows from one of three proven induction theorems, which have recently been proven using the above definition of R_{close}. But the second induction principle used in [CM88] still has to be proven correct or rejected.

Finally, some lattice theoretical results, not presented here, proving R_{close} being the least fixpoint of $RelFam$ have been achieved.

References

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design, A Foundation.* Addison Wesley, 1988.

[Coh76] P. M. Cohn. *UNIVERSAL ALGEBRA.* Springer Verlag, 1976.

[Coo69] D.C. Cooper. Program scheme equivalences and second-order logic. *Machine Intelligence*, 4:3–15, 1969.

[Gordon88] M. J. Gordon. *HOL: A proof generating system for higher order logic.* in *VLSI Specification, Verification and Synthesis*, G. Birtwistle and P. A. Subrahmanyam, Eds. Boston, MA: Kluwer Academic, 1988, pp. 73–128.

[Melham88] T. F. Melham. *Automating Recursive Type Definitions in Higher Order Logic.* Technical Report No. 146, Computer Laboratory, The University of Cambridge, September 1988.

[Par69] David Park. Fixpoint induction and proofs of program properties. *Machine Intelligence*, 5:59–78, 1969.

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

11.30-12.00
Solving Reflexive Domain Equations using HOL

Kim Dam Petersen
TFL (A Telecomms Research Laboratory)

Introduction

The type package made by Tom Melham [Melham] allows for defining recursive data structures of the form:

$$T = v_1\,\tau_1^1\,\ldots\,\tau_1^{m_1}\;|\;\ldots\;|\;v_n\,\tau_n^1\,\ldots\,\tau_n^{m_n},$$

where each $v_i,\ 1 \le i \le n$ is a variant label, and each $\tau_i^j,\ 1 \le i \le n,\ 1 \le j \le m_i$ is either an already defined type or a reference to the data structure T being defined. The class of data structures definable by this package may be characterized as "tree" like structures. This class of data structures unfortunately excludes the various kinds of procedure denotations.

An example of a recursive data structure, that cannot be defined using the type package is:

$$C = I \to C \to O,$$

which defines a class of computers C that given an instruction of type I and a computer of it's own kind (C) produces an output result of type O.

We would like a type package that is able to solve any general recursive data structure, i.e. structures of the form:

$$T = \mathcal{E}(T),$$

where $\mathcal{E}(T)$ is a general type expression in T. An Equation of this general form will be called a Reflexive Domain Equation (RDE).

Solving Reflexive Domain Equations

A method of how RDEs may be solved using ordinary mathematics is given in [Schmidt]. This method solves an RDE of the form: $D = \mathcal{E}(D)$ by first defining a sequence of domains D_i based on the RDE:

$$D_0 = \{\perp\}, D_{n+1} = \mathcal{E}(D_n),$$

and then of proving that this sequence "converge" to a domain D_∞, which is isomorphic to $\mathcal{E}(D_\infty)$.

The intention is to use this method to solve RDEs in HOL. In order to represent in HOL the concepts used by the method, it is necessary to have an object called *Domain* that represents all possible domains, i.e., an object that includes the domain of numbers and boolean, and which is closed wrt. Pair, Union and Function construction. The set of closed type free λ-expression constitutes such an object, hence if we can represent closed type free λ-expression we can solve RDEs in HOL. Looking into [Barendregt] we find that any reflexive cpo can be used to construct a representation of closed type free λ-expressions.

A reflexive cpo is a cpo $(D, \sqsubseteq)$ for which two continuous functions $f : D \to [D \to D]$ and $g : [D \to D] \to D$ exists such that: $f \circ g = id_{[D\to D]}$. If we consider g as an encoding function for continuous function from D to D and f as the corresponding decoding function, we may characterize a reflexive cpo as a cpo in which the continuous function from D to D may be encoded as objects of D.

Furthermore [Barendregt] proves that $(P^\omega, \subseteq)$, in which $P^\omega = \{x | x \subseteq \mathrm{N}\}$ is a cpo, and show how to construct functions f and g such that P^ω becomes a reflexive cpo. The construction of continuous functions f and g is explained in the following. First pairs of numbers are uniquely encoded by numbers as:

$$[n, m] \leftrightarrow \frac{1}{2}(n+m)(n+m+1)+m,$$

and finite subsets of numbers are uniquely encoded by numbers as:

$$\{k_0, ..., k_{m-1}\} \leftrightarrow e_n, n = \sum_{0 \leq i < m} 2^{k_i}.$$

Using these encodings the (*graph*) function g may be defined as:

$$g = (f) \mapsto \{[n, m] | m \in f(e_n)\} : [[P^\omega \to P^\omega] \to P^\omega],$$

and the (*fun*) function f as:

$$f = (u)(x) \mapsto \{m | \exists e_n \subseteq x : [n, m] \in u\} : [P^\omega \to [P^\omega \to P^\omega]].$$

The property $f \circ g = id_{[P^\omega \to P^\omega]}$ is then proved.

Conclusions

That P^ω may be organized as a reflexive cpo has been proved in HOL. Representing type free λ calculus has just been started.

Acknowledgements

During the presentation of this work at the 3'rd HOL User's meeting 1990 in Århus, Michael Gordon and Glynn Winskel pointed out that RDEs may be solved directly from P^ω without using type free λ calculus. How to do this is described in [Scott].

References

[Melham] T. F. Melham. *Automating Recursive Type Definitions in Higher Order Logic.* Technical Report No. 146, Computer Laboratory, The University of Cambridge, September 1988.

[Schmidt] D. A. Schmidt. *Denotational Semantics.* Allyn and Bacon, 1986.

[Barendregt] H. P. Barendregt. *The Lambda Calculus.* North Holland, 1984.

[Scott] D. Scott. *Data types as lattices.* in SIAM J. COMPUT. Vol. 5, No. 3, September 1976.

12.00-12.30
Towards a Theory for Finite Word Length Arithmetic

Wim Ploegaerts[1]

Imec vzw., Kapeldreef 75, 3003 Leuven, Belgium

e-mail: `ploegaer@imec.be`

L. Claesen, H. De Man

Professor at the K.U.Leuven

The main goal of the ongoing research in our group is the development of a *silicon compiler* for DSP (Digital Signal Processing) applications, CATHEDRAL. "*... A silicon compiler is a CAD environment that realizes the synthesis of an algorithmic description of an application and a number of design constraints, to the final collection of polygons* [the layout], *to be delivered to a* (silicon)*foundry ... CATHEDRAL-II is a synthesis system for real-time, complex decision-making signal processing applications. The generated chips perform complex arithmetic operations ...*" [1].

Within the CHEOPS Basic Research Action, a EC project between the Computer Lab Cambridge, Imec and Philips, we are investigating whether formal verification offers a valuable strategy for (part of) the validation of the synthesized circuits. In this context "valuable" is determined by the needs of the synthesis process, implementing the concept of going "*... from an idea to layout in one day ...* ". The implementation of dedicated decision procedures which automate part of the verification work will therefore be essential.

As a first step it has been tried to model the arithmetic operations, which are typically finite wordlength operations on bitstrings using a two's complement representation. To be able to use the concepts and benefits from the abstraction methodology [2], a link from bitstrings to integer numbers had to be provided. In [3] a very detailed description of the denotation and representation functions for both signed and unsigned arithmetic is provided. The implementation of these functions in HOL urged to a vast extension of the existing theories on integer numbers. Besides that, some more list theory had to be provided. The conceptual extensions which have been made are

- definition of the "inverse" of HD and TL such that lists can be treated as "symmetric objects" (i.e. LAST, the sublist without the last element), including the related definitional scheme
- definition of the absolute value, integer division and remainder (based on Euclids theorem)
- definition of "two", the concept of even and odd integers, log2, power of "two"
- implementation of a decision procedure to proof that two integer expressions (containing "+", "-", "*", variables, functions and constants) are equal

Using these concepts, the signed denotation and representation function, mapping bitstrings on integers and the other way round, has been defined.

---

[1]Research Assistent with the Belgian National Fund for Scientific Research

Work partly funded by the ESPRIT-2 BRA CHEOPS (3215)

All extensions of “zet” and “list” are documented and available (and can be found in the contrib directory of the next release), including documentation on “zet” itself and on some of the tactics and conversions defined in “auxiliary”. The actual implementation of [3] still has to be documented.

References

[1] I.Bolsens: “Definition of Silicon Compilers and their Evaluation Criteria”, Imec int.rep. IM55270-1, Issue 02.00, June 1990

[2] T.F.Melham: “Abstraction Mechanisms for Hardware Verification”

[3] R.T.Boute: “Denotational and Representational Semantics of Digital Systems”, K.U.Nijmegen int.rep.

Day 2, Afternoon Session

14.00-15.00
More Reasons Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware

Jeffrey J. Joyce

Department of Computer Science
University of British Columbia
Vancouver, B.C., CANADA V6T 1W5

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 Mike Gordon's 1985 paper, "Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware". This talk (based on [3]) proposes reasons 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.

In addition to the well known advantages of modularity, abstraction and reliable reusability, the use of generic specification to eliminate non-essential detail from a formal specification sharpens the distinction between what has and what has not been formally considered. In a hierarchical proof effort, the elimination of non-essential detail isolates each level from details only relevant to other levels. The elimination of non-essential detail also reduces the need for special-purpose proof infrastructure for reasoning about hardware, e.g., hardware-oriented data types. Elsewhere [1, 2], we have described a technique for writing generic HOL specifications based on the use of higher-order predicates parameterized by function variables and type variables. This technique avoids extensions to the HOL logic (for expressing genericity) which may unnecessarily complicate the HOL logic.

A second reason for claiming that higher-order logic is a good formalism for specifying and verifying hardware is the ability to embed special-purpose formalisms in higher-order logic. Diverse aspects of hardware behaviour are described informally in an equally diverse range of styles: finite-state machines, pseudo-code, logic expressions, timing diagrams are some common examples. Conventional hardware specifications attempt, in general, to recast these diverse forms of informal description into a fixed-format (e.g., recursive function definitions). However, we advocate a different approach, namely, to embed *natural notations* from well-established formalisms such as temporal logic in the framework of higher-order logic. This approach benefits from the built-in economy of these special-purpose notations when they are applied to particular areas of formal description. These benefits include improved specifications (more concise and easier to read) and ease of proof (more powerful proof rules).

References

[1] Jeffrey J. Joyce, Multi-Level Verification of Microprocessor-Based Systems, Ph.D. Thesis, Computer Laboratory, Cambridge University, December 1989. Report No. 195, Computer Laboratory, Cambridge University, May 1990.

[2] Jeffrey J. Joyce, Generic Specification of Digital Hardware, in: M. Sheeran and G. Jones, eds., Proceedings of Workshop on Digital Circuit Correctness, September 1990, Oxford.

[3] Jeffrey J. Joyce, More Reasons Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware, in: 1991 International Workshop on Formal Methods in VLSI Design, 9-11 January 1991, Miami, Florida.

15.00-15.30
A Low-level Circuit Model in HOL
Aarhus students

16.00-16.30
The Verification of a Parameterized Mead and Conway Alu Core in HOL
Catia Marcondes Angelo, Luc Claesen, Hugo De Man
IMEC, Belgium

In a silicon compiler environment as CATHEDRAL, hardware is synthesised by means of transformations through many levels of abstraction starting from a high level specification down to a layout. Since the synthesis is made using a library of parameterized modules as building blocks, their correctness is crucial.

This work concerns about the development of a methodology to formally verify parameterized modules using HOL. The research is based on the verification of the parameterized Mead & Conway alu core implemented in the CATHEDRAL II system.

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.

From that work we concluded that the reasoning of the designer is essential in the verification process and removing irregularities from the implementation before going to the specification level is a powerful strategy to structure the correctness proof. Moreover it is possible to build a hardware verification environment with useful general theorems and tactics exploring abstractions in Higher Order Logic.

16.30-17.00

The HOL Verification of ELLA Designs

Richard J. Boulton, Mike Gordon, John Herbert, John Van Tassel
University of Cambridge Computer Laboratory
New Museums Site, Pembroke Street
Cambridge CB2 3QG, England
Email: rjb@cl.cam.ac.uk Phone: +44 223 334729 Fax: +44 223 334678

Two methods currently used to specify hardware are

1. hardware description languages, e.g. ELLA, VHDL, and

2. formal systems, e.g. HOL, Boyer-Moore logic.

There is currently a gap between these two methods. We aim to build a bridge between the two methods for ELLA and the HOL proof assistant. Specifically, we want to make formal specification and proof available within a conventional design process, and provide access to the engineering tools (*e.g.* the simulator) in a theorem proving environment. Our approach is to *semantically embed* a subset of ELLA in higher order logic.

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 and the ability to pretty-print the HOL terms as the original ELLA constructs. The simplicity of the translation means that it can readily be seen to be correct, and the output from the pretty-printer confirms this, as the output will be almost identical to the original ELLA text. 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 HOL-ELLA system has a parser which parses ELLA text to a parse-tree held as an ML datastructure. This parse-tree can then be used to compute the HOL term representing the semantics of the ELLA text. It is possible for the translation to fail due to syntax errors not expressed by the grammar or due to semantic errors. In both cases, the offending part of the parse-tree is pretty-printed together with a suitable error message. The HOL term representing the semantics can also be pretty-printed to appear as the original ELLA text.

As a simple example, a circuit consisting of a single inverter can be specified by the ELLA *series*

```
BEGIN
   MAKE INV: out.
   JOIN in -> out.
OUTPUT out
END
```

The HOL representation of this ELLA is

```
@OUTPUT.
 ?out out_fn.
```

```
SERIES
[MAKE[[MAKEITEM INV out_fn]];
 JOIN
 [JOINITEM in out out_fn (\x'. @f1'. !t'. f1' t' = x' t')]]
OUTPUT
out
```

The existentially quantified variable `out_fn` is required in addition to `out` so that each `MAKEITEM` and `JOINITEM` can have the same type. This allows them to be placed in lists. The final argument to the semantic constant `JOINITEM` is a type-casting function. It takes a signal whose value is a tuple, and yields a tuple of signals. In the example, the tuple is a 1-tuple, so the function does nothing. Simplifying this function and eliminating the semantic constants gives the term:

```
@OUTPUT.
 ?out out_fn.
  (out_fn = INV) /\
  (out_fn in = out) /\
  (OUTPUT = out)
```

This can be simplified further to yield

```
INV in
```

The simplification can be done automatically.

Another interesting feature of ELLA are *constants*. These are used as initial values for devices with state and as 'patterns' in the CASE statement. The ELLA CASE statement behaves as a multiplexer, controlled by matching the input value against patterns (*constants*). There is only one syntactic category for constants. A given constant may make sense in only one of the two roles. In order to keep the ELLA to HOL translation context-free, constants are translated in the same way whichever role they take on. Constants are modelled as predicates. This is consistent with their use as patterns. For initial values, Hilbert's ε-operator is used to obtain a value from the predicate (but only if the predicate specifies a unique value).

There is a problem with the semantics of constants when used as patterns. ELLA features an unspecified value denoted by `?`. `?` may be generated by arithmetic operations producing a result out of range, by CASE statements with *choosers* (ELLA constants) which are not exhaustive, or explicitly. `?` represents one of the possible values of the appropriate type, but it is not known which. 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 example, for an ELLA type `bit` with the two possible values `hi` and `lo`

```
hi | lo
```

is a constant which 'matches' `hi` or `lo`, but *not* `?`. However,

```
bit
```

is a constant which 'matches' `hi` or `lo` or `?`.

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"`. This uniform approach was chosen in preference to explicitly adding an extra element to every HOL type used to represent an ELLA type. In addition, lifted Boolean values are useful for describing the semantics of the CASE statement.

In conclusion, 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 in proofs, so we may produce a new version of the semantics which does not involve lifting but which does not quite conform to the semantics of ELLA. The subset chosen is large. Further research is required to determine which parts could reasonably be discarded and whether any necessary ELLA constructs have been omitted. We have also produced some tools for simplifying the machine-generated terms. These could probably be improved to provide more automation.

The bulk of future work will be concerned with design and verification case-studies (e.g. a UART and a small microprocessor). By doing these, we hope to develop a methodology for using the combined HOL-ELLA system.

A more detailed description of this work can be found in University of Cambridge Computer Laboratory Technical Report number 199.

17.00-17.30
HOL Semantics of SILAGE

Ton Kalker,
Philips Research Laboratories, The Netherlands.

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.