CS745/ECE-725: Computer-Aided Verification

Topic 3: Mechanized Interactive Theorem Proving

http://www.student.cs.uwaterloo.ca/~cs745
uw.cs.cs745
cs745@student.cs.uwaterloo.ca
Nancy Day

Outline

- What is mechanized theorem proving?
- Demo
- Overview and history
- Background
  - Lambda calculus
  - ML programming language
- HOL theorem proving system

Theorem Proving

"Automated deduction or theorem proving refers to the mechanization of deduction reasoning." [R8], p. 79

Advantages:

- automatic proof generation when possible
- bookkeeping
- proof checking
- user extensibility
- expressiveness of the logic

"Mechanized" means using a software tool.

Theorem Proving

Disadvantages:

- automatic proof generation is only possible for a limited range of logics (decidability)
- not allowed to say "and obviously ..." as a proof step – computers require all the proof steps to be spelled out

What is a theorem prover?

In its essence,

A theorem prover is a pattern matching tool.

1-demo.sml

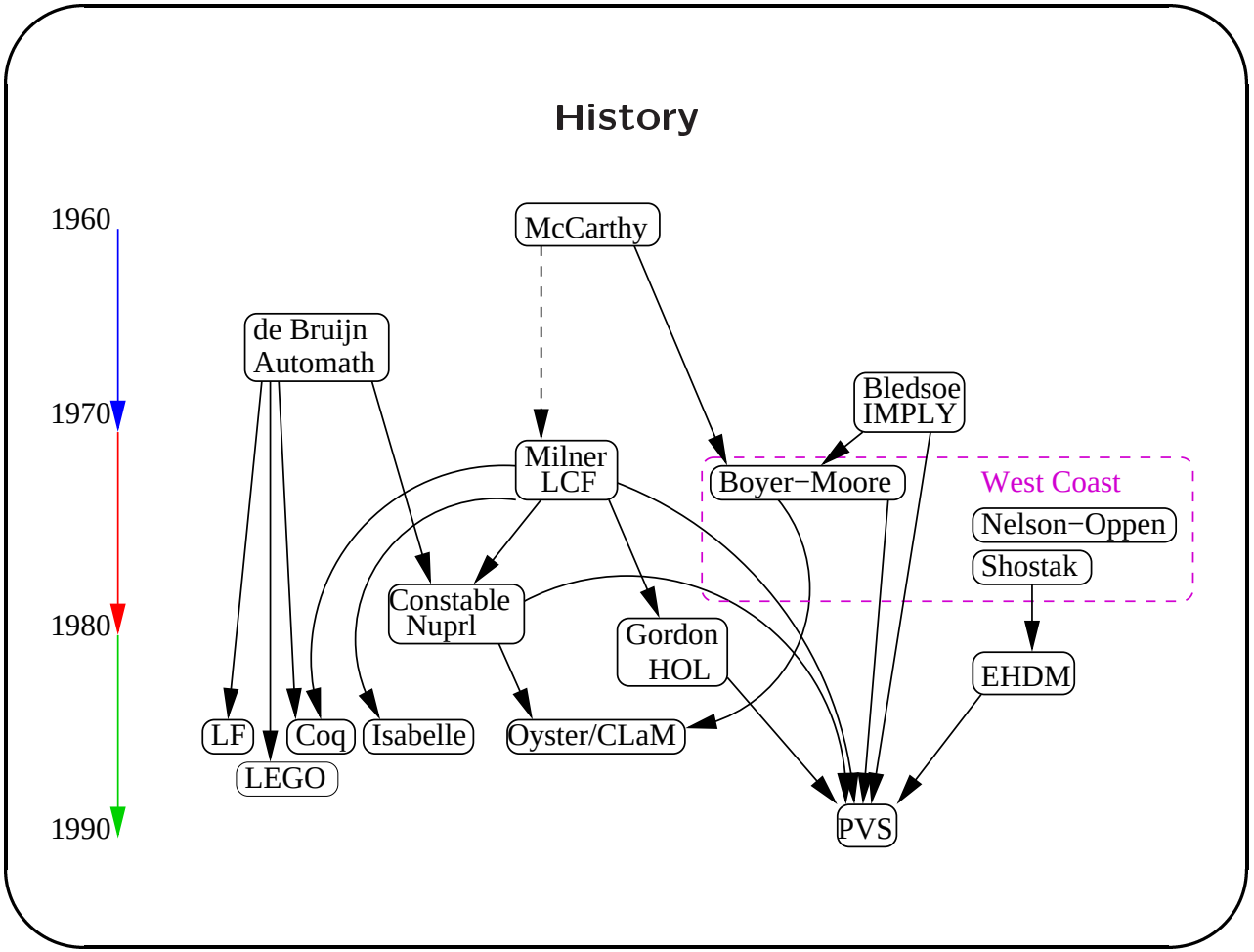
Theorem proving: Overview

Theorem proving techniques and tools range from fully automatic to just proof checking.

In between these two extremes, tools have:

- derived proof rules and smart pattern matching (e.g., HOL)
- discovery of induction invariants (e.g., ACL2)
- fully automatic proof procedures with "flags" (e.g., Otter)
- minimalist proof assistants to model checkers (e.g., SMV)
- combined model checkers and theorem provers

Historical Perspective



Src: N. Shankar's invited talk at TPHOLs 2001

See also: Donald MacKenzie, *Mechanizing Proof Computing, Risk, and Trust*, MIT Press, 2001

LCF-style mechanical theorem proving

The HOL system is an example of LCF-style mechanical theorem proving.

LCF-style features:

- separation of object language (logic) and meta-language
- meta-language is a typed functional language (ML)
- theorem is an abstract datatype; only way to construct theorems is using the axioms and inference rules
- "secure" theorem proving achieved by having a small core; everything else is built on top of that core
- backward proof (goal directed proof): tactics, tacticals
- programming in the meta-language is used to create bigger combinations of steps

History

- 1972: Milner's Stanford LCF (Logic of Computable Functions)
  - mechanize proofs in PPLAMBDA, chosen because it was appropriate for the study of semantics of programming languages
  - ways of creating subgoals, storing theorems to build a library
  - highly repetitive work

History

- 1977: Edinburgh LCF (Milner et al.)
  - introduction of a **metalanguage** (ML):
    - could encode not just primitive reasoning steps, but derived rules
    - use of a **type system** to distinguish between "terms" and "theorems"; theorems could only be created using the axioms and inference rules. Therefore the soundness of the tool depends only on the primitive axioms and rules of inference, and the type checking code.
    - interface of the logic to the meta-language explicit using the type structure of ML, so that other logics could be tried

History

- ported from Stanford Lisp to Franz Lisp by Gérard Huet at INRIA
- Cambridge LCF: further developed at Cambridge by Larry Paulson
- HOL: Gordon [R5], [R4]
  - implemented on top of Cambridge LCF
  - for classical higher order logic (Church's simple theory of types [R2])
  - originally higher order logic was chosen to be suitable for hardware verification although it has many other applications now

History

- HOL88 (1988)
- HOL90: port of HOL88 to SML by Konrad Slind
- HOL98: incorporation of many decision procedures (some as HOL derived rules)
- HOL 4: latest version (2004-)

Using HOL

Source/Installer for HOL: http://hol.sourceforge.net/ (You have to insall Mosml 2.01 first.)

You can use HOL on the WatForm servers (tumbo, quadra, mudge, gooch) or you can install HOL yourself. On the WatForm servers, it is in watform/bin.

Using HOL

To run HOL (on Windows): hol.bat

```
Moscow ML version 2.01 (January 2004)
Enter ‘quit();’ to quit.
[opening file "C:\Program Files (x86)\Hol\std.prelude"]

------------------------------------------------------------
       HOL-4 [Kananaskis 5 (built Wed Jul 08 21:08:36 2009)]

       For introductory HOL help, type: help "hol";
------------------------------------------------------------

[loading theories and proof tools .............. ]
...
- ...

- ^D      (to exit)
```

Using HOL

The reference manuals for HOL are all on-line at the HOL web site. The course web page contains links to several useful references on HOL and ML.

HOL has a command line interface. It is convenient to interact with HOL through emacs. There are instructions on the "References" course web page for how to set up the emacs interface. An alternative is to cut and paste between your proof script window and the HOL session.

Outline

- What is mechanized theorem proving?
- Demo
- Overview and history
- Background
  - Lambda calculus
  - ML programming language
- HOL theorem proving system

Lambda Calculus: Background

Src: Gordon [R5]

The lambda calculus is a theory of functions developed by Alonzo Church in the 1930s in an attempt to create a formal representation of computation.

Church-Turing Thesis: all of the following are formally equivalent and represent the "effectively calculable functions",

- lambda calculus
- Turing machines
- *and many others*

Influence of the Lambda Calculus

The lambda calculus inspired:

- LISP, McCarthy, 1950's
- ISWIN, Landin, 1960 (introduced main notations of functional programming)
- Denotational semantics, Strachey
- Domain theory, Scott
- functional programming and techniques for its implementation

Lambda Calculus: Syntax

The lambda calculus is a notation for defining functions. The expressions of the language denote functions.

$$\begin{array}{rcl} \langle \lambda\text{-expr} \rangle & ::= & \langle \textit{variable} \rangle \\ & | & \lambda \langle \textit{variable} \rangle . \ \langle \lambda\text{-expr} \rangle \\ & | & \langle \lambda\text{-expr} \rangle \ \langle \lambda\text{-expr} \rangle \end{array}$$

An abstraction is an expression of the form $\lambda\langle \textit{variable} \rangle . \langle \lambda\text{-expr} \rangle$. In an abstraction, $\lambda x.e$, we call x the bound variable and e is the body.

An application is an expression of the form $\langle \lambda\text{-expr} \rangle \ \langle \lambda\text{-expr} \rangle$. In an application, $e_1 \ e_2$, we call e_1 the rator (from operator), and e_2 is called the rand (from operand).

Abstractions

An abstraction $\lambda x.e$ denotes the function that takes an argument a and returns the function denoted by the body e, in an environment where x denotes a.

$$(\lambda x.\, e)\; a \;\equiv\; e[a/x]$$

An abstraction is an anonymous function. The two definitions below are equivalent.

$$\begin{aligned} f\; x &= x+1 \\ f &= \lambda x.\, x+1 \end{aligned}$$

Examples

Examples of syntactically legal lambda expressions:

$\lambda x.\, x$	identity function
$\lambda x.\, x + 1$	increment function
$(\lambda x.\, x + 1)\ 3$	increment function applied to 3
$(\lambda x.\, x + 1)\ ((\lambda x.\, x + 1)\ 2)$	inc applied to the inc of 2

Examples of syntactically illegal lambda expressions:

$\lambda(x + 1).\, x + 1$	bound variable must be a variable, not a term

Conventions

1. function application associates to the left:

$$e_1\ e_2\ e_3\ e_4 \equiv (((e_1\ e_2)\ e_3)\ e_4)$$

2. the scope of $\lambda x. \ldots$ extends as far right as possible

$$\lambda x.\ e_1\ e_2\ e_3 \ldots e_n \equiv (\lambda x.\ (e_1\ e_2\ e_3 \ldots e_n))$$

3. nesting of abstractions

$$\lambda x_1 \ldots x_n.\ e \equiv (\lambda x_1.\ (\ldots (\lambda x_n.\ e)))$$

Example: $\lambda x\ y.\ f\ y\ x \equiv (\lambda x.\ (\lambda y.\ ((f\ y)\ x)))$

Bound and Free Variables

An occurrence of a variable x is free if it is not within the scope of a λx, otherwise it is bound.

This is the same definition of free and bound that we had previously for quantifiers.

Lambda Calculus: Calculating

There are three rules (conversions/reductions) for calculating with the lambda calculus. These rules allow us to simplify or reduce lambda expressions.

Alpha (α)	rename bound variables
Beta (β)	reduce / simplify expressions with function application
Eta (η)	equivalent representations of functions

Substitution

The calculations involve substitution. As in predicate logic, we have to avoid variable capture.

$e[e'/x]$ means substituting e' for each free occurrence of x in e. The substitution is valid if no free variable in e' becomes bound in the result.

Next, we define substitution such that we no longer have to worry about variable capture.

Substitution

	e	$e[e'/v]$
1	v	e'
2	v' (where $v \neq v'$)	v'
3	$e_1\ e_2$	$(e_1[e'/v])\,(e_2[e'/v])$
4	$\lambda v.\ e_1$	$\lambda v.\ e_1$
5	$\lambda v'.\ e_1$ (where $v \neq v'$ and v' **is not free** in e')	$\lambda v'.\ (e_1[e'/v])$
6	$\lambda v'.\ e_1$ (where $v \neq v'$ and v' **is free** in e')	$\lambda v''.\ (e_1[v''/v'])[e'/v]$ where v'' is a variable not free in e' or e_1

We will assume we are using this substitution in all subsequent slides.

Example of Substitution

$(\lambda y.\, y\ x)\,[y/x]$ $\quad$ y is free in $y\ x$

$$
\begin{aligned}
(\lambda y.\, y\ x)\,[y/x] &= \lambda y''.\, ((y\ x)\,[y''/y])[y/x] \\
&= \lambda y''.\, ((y\ [y''/y])\ (x\ [y''/y]))[y/x] \\
&= \lambda y''.\, (y''\ x)[y/x] \\
&= \lambda y''.\, (y''[y/x])\ (x[y/x]) \\
&= \lambda y''.\, y''\ y
\end{aligned}
$$

Alpha Conversions

Alpha conversion: renaming bound variables in abstractions
$\lambda x.\, e \underset{\alpha}{\mapsto} \lambda x'.\, e[x'/x]$

legal α-conversion replacing a with b:

$$\lambda a.\, (a+1) \quad \underset{\alpha}{\mapsto} \quad \lambda b.\, (b+1)$$

illegal α-conversion :

$$(a+1) \quad \underset{\alpha}{\mapsto} \quad (b+1) \quad \text{“a” is a free variable, so can't be renamed}$$

Beta Conversions

Beta conversion: simplifying function application

$(\lambda x.\, e_1)\; e_2 \underset{\beta}{\mapsto} e_1[e_2/x]$

Example:

$$(\lambda a.\, (a+1))\; 3 \underset{\beta}{\mapsto} 3+1$$

Example

What does the following function do?

$$\lambda x.\, \lambda y.\, y$$

Eta Conversions

Two functions are the same if they give the same result when applied to the same arguments. This property is called extensionality.

The eta conversion is $\lambda x.\,(e\ x) \underset{\eta}{\mapsto} e$ provided x has no free occurrence in e.

Example: $\lambda x.\, f\ 5\ x \underset{\eta}{\mapsto} f\ 5$

Generalization and Equality

$\underset{\alpha}{\mapsto}, \underset{\beta}{\mapsto}, \underset{\eta}{\mapsto}$ are also used to describe the application of the conversion to any **subterm**. We use $\mapsto$ to describe the application of one of the conversions.

These rules preserve the meaning of the expression, i.e., the resulting expression denotes the same function. We say that two expressions are equal ($=$) if they can be transformed into each other by a sequence of conversions.

Extensionality

Recall Leibniz's Law:
If $t_1 = t_2$ is a theorem, then so is $P[t_1/x] \Leftrightarrow P[t_2/x]$.

If v is not free in e_1 or e_2, and $e_1\ v = e_2\ v$

then by Leibniz's law, $\lambda v.\ e_1\ v = \lambda v.\ e_2\ v$

and by η-conversion on both sides $e_1 = e_2$

So to prove two λ-expressions are equal ($e_1 = e_2$), we can prove $e_1\ v = e_2\ v$ for some v not occurring free in e_1 or e_2. This is called proof by extensionality.

Church-Rosser Theorem

Church-Rosser Theorem: if an expression e_0 can be reduced by zero or more reduction steps to either expression e_1 or e_2, then there exists an e such that $e_1 \mapsto e$ and $e_2 \mapsto e$.

$$
\begin{array}{rl}
 & (\lambda \underline{a}.\ ((\lambda b.\ b\ a)\ (\lambda x.\ x+1)))\underline{3} \\
= & (\lambda \underline{b}.\ b\ \mathbf{3})\ \underline{(\lambda x.\ x+1)} \\
= & (\lambda \underline{x}.\ x+1)\ \underline{3} \\
= & 3+1
\end{array}
\qquad
\begin{array}{rl}
 & (\lambda a.\ ((\lambda \underline{b}.\ b\ a)\ \underline{(\lambda x.\ x+1)}))\ 3 \\
= & (\lambda \underline{a}.\ ((\lambda \mathbf{x}.\ \mathbf{x}+\mathbf{1})\ a))\ \underline{3} \\
= & (\lambda \underline{x}.\ x+1)\ \underline{\mathbf{3}} \\
= & \mathbf{3}+1
\end{array}
$$

Basically, the order of beta reductions doesn't matter.

Aside: Eager vs Lazy Evaluation

For beta reduction, there are two common ways to carry out the reduction.

Eager evaluation: evaluate arguments before substitution

Lazy evaluation: delay evaluation of arguments until needed

Lazy		Eager	
	$(\lambda \underline{x}.\, x+1)\ \underline{((\lambda y.\, 2 \times y)\ 3)}$		$(\lambda x.\, x+1)\ ((\lambda \underline{y}.\, 2 \times y)\ \underline{3})$
$=$	$((\lambda \underline{\mathbf{y}}.\, \mathbf{2} \times \mathbf{y})\ \underline{\mathbf{3}}) + 1$	$=$	$(\lambda \underline{x}.\, x+1)\ \underline{(\mathbf{2} \times \mathbf{3})}$
$=$	$(\mathbf{2} \times \mathbf{3}) + 1$	$=$	$(\lambda \underline{x}.\, x+1)\ 6$
$=$	$\mathbf{6} + 1$	$=$	$\mathbf{6} + 1$
$=$	7	$=$	7

Aside: Eager vs Lazy Evaluation

Lazy evaluation can sometimes avoid non-termination (Ω):

	Lazy	Eager
	$(\lambda \underline{x}.\ ((\lambda y.\ y)\ 5))\ \underline{\Omega}$	$(\lambda x.\ ((\lambda y.\ y)\ 5))\ \underline{\Omega}$
$=$	$(\lambda y.\ y)\ 5$	evaluates to Ω
$=$	5	which never terminates

Aside: Eager and Lazy Evaluation

A naive implementation of lazy evaluation can be inefficient if arguments are used multiple times in a function body.

Lazy		Eager	
	$(\lambda \underline{x}.\, x + x)\ \underline{((\lambda y.\, 2 * y)\ 3)}$		
$=$	$((\lambda \underline{y}.\, 2 * y)\ \underline{3}) + ((\lambda y.\, 2 * y)\ 3)$		$(\lambda x.\, x + x)\ \underline{((\lambda y.\, 2 * y)\ 3)}$
$=$	$\underline{(\mathbf{2} * \mathbf{3})} + ((\lambda y.\, 2 * y)\ 3)$	$=$	$(\lambda \underline{x}.\, x + x)\ \underline{(\mathbf{2} * \mathbf{3})}$
$=$	$6 + ((\lambda \underline{y}.\, 2 * y)\ \underline{3})$	$=$	$(\lambda \underline{x}.\, x + x)\ \underline{6}$
$=$	$6 + \underline{(\mathbf{2} * \mathbf{3})}$	$=$	$6 + 6$
$=$	$6 + 6$		

Most implementations of lazy evaluation use term-sharing representations so that evaluation of $(\lambda y.\, 2 * y)\ 3$ is done only once.

Representing Boolean Operators

We can encode the objects **true** and **false** and the Boolean operators as λ-expressions that have the right properties.

$\mathsf{true} \equiv \lambda x.\, \lambda y.\, x$
$\mathsf{false} \equiv \lambda x.\, \lambda y.\, y$
$\mathsf{not} \equiv \lambda t.\, t\ \mathsf{false}\ \mathsf{true}$

These definitions have the appropriate properties:

$$\begin{aligned} \mathsf{not\ true} &= (\lambda t.\, t\ \mathsf{false}\ \mathsf{true})\ \mathsf{true} \\ &= \mathsf{true}\ \mathsf{false}\ \mathsf{true} \\ &= (\lambda x.\, \lambda y.\, x)\ \mathsf{false}\ \mathsf{true} \\ &= (\lambda y.\, \mathsf{false})\ \mathsf{true} \\ &= \mathsf{false} \end{aligned}$$

We can do the same kind of coding to represent pairs and numbers, etc.

Functions with Several Arguments

In mathematics, we're used to writing $f(x_1, \ldots, x_n)$ for the application the n-ary function f to the arguments $x_1, \ldots, x_n$.

In the λ-calculus, there are two ways to write such a function:

1. curried: $f\ x_1 \ldots x_n$
   f expects its arguments "one at a time"
2. uncurried: $f(x_1, \ldots, x_n)$
   f expects its arguments all at the same time

A curried function can be partially applied.

Lambda Calculus, ML, and HOL

ML and higher order logic both use the lambda calculus to represent functions.

ML is the meta-language for the theorem prover HOL.

Higher order logic is the object language (logic) for HOL.

Outline

- What is mechanized theorem proving?
- Demo
- Overview and history
- Background
  - Lambda calculus
  - ML programming language
- HOL theorem proving system

What is a functional language?

ML is an eager functional language.

In an imperative programming language there are expressions and commands. Expressions evaluate to a value and commands change the state of the computer.

In a functional program, everything is a expression. There are no commands. The whole program is a single function.

Instead of loops, functional programs use recursion.

Functions can take other functions as arguments. These are called higher order functions.

There are no pointers. The data structures used are mostly lists and trees.

See 2-ml-demo.sml

Types

ML is a strongly-typed language, with polymorphic typechecking.

> *strongly typed: strict enforcement of type rules with no exceptions. All types are known at compile time, i.e. are statically bound.*

A polymorphic function is one that can be applied to an argument of any type.

Often, ML can infer all the types involved in an expression, so you don't have to provide explicitly typing information.

See 2-ml-demo.sml

Referential Transparency

Generally, functional programs satisfy the property of referential transparency: we can replace a subexpression with another expression that is equal to it. The context of the expression doesn't matter.

When is $x + y \neq y + x$?

Outline

- What is mechanized theorem proving?
- Demo
- Overview and history
- Background
  - Lambda calculus
  - ML programming language
- HOL theorem proving system

Outline

The HOL theorem proving system:

- Higher order logic
- Features of LCF-style theorem provers:
  - derived rules
  - theories
  - backward proof (tactics, tacticals)
- Examples
- HOL hammers
- Writing tactics
- Harry Potter
- DOIT_TAC

The HOL system

The HOL theorem proving system is written in ML.

- the logic is represented as a datatype in ML
- formulae in the logic are instances of this datatype (placed in single quotes and called 'terms')
- ML functions manipulate terms in the logic (rules of inference, tactics)

Object Language and Meta-Language

ML manipulates expressions in the logic.

Expressions in the logic have ML type "term".

Forward proof steps (inference rules), and backward proof steps (tactics) are functions in ML.

A theorem (ML type "thm") is pair that consists of

1. a list of terms (assumptions)
2. a single term (conclusion)

An inference rule can take a term and produce a theorem. It may also take other theorems as arguments.

Single Quotes

- Expressions in single back quotes (e.g., ‘!x. P(x)‘) have type “term frag list” in ML.

Inference rules (implemented as ML functions) take term frag lists.

Antiquotation means that you can use an ML identifier in the middle of a term frag list and it will replace it with the term associated with the ML identifier.

(Occasionally you will see an old function that wants a “term” rather than a “term frag list” as an argument. In this case, put the term in double quotes as in “!x. P(x)“.

Higher Order Logic

Recall that in first order logic, quantifiers can only range over variables. In higher order logics, functions can take functions as arguments, and quantifiers can range over functions.

This is because higher order logic uses lambda terms (λ-terms) to describe functions, a much more expressive way to describe functions than is available in first order logic.

There are multiple kinds of higher order logic. The higher order logic used in the HOL theorem proving system is a variant of Church's simple theory of types [R2]. From now on we will be refering to the flavour used in the HOL theorem proving system. Other flavours include those used in PVS [R9], IMPS [R3], Nuprl [R1], Coq [R6], and Isabelle [R10].

Logic Syntax

Standard Notation	HOL notation
true	`T`
false	`F`
$\neg t$	`~t`
$t_1 \wedge t_2$	`t1 /\ t2`
$t_1 \vee t_2$	`t1 \/ t2`
$t_1 \Rightarrow t_2$	`t1 ==> t2`
$t_1 = t_2$	`t1 = t2`

Logic Syntax

Standard Notation	HOL notation
$\forall x.t$	`!x.t`
$\exists x.t$	`?x.t`
$\varepsilon x.t$	`@x.t`
$t \longrightarrow t_1 \mid t_2$	`t => t1 | t2`
$\lambda x.t$	`\x. t`

$\varepsilon x.t$ means an x such that t. ε is called Hilbert's choice operator.

Logic Terms

There are four kinds of terms in higher order logic:

1. Variables, which can be bound by quantifiers
2. Constants
3. Function applications: `t1 t2`
4. Lambda-abstractions: `\x. t`

Notice that it uses a different notation for λ than ML.

Identifiers in HOL are case sensitive.

See 3-hol-basics.sml

Russell's Paradox

When functions are allowed to take other functions as arguments, we can run into trouble with paradoxes. The following is a version of Russell's paradox:

$$P\ x = \neg(x\ x)$$

From which we can conclude: $P\ P = \neg(P\ P)$

Paradox: A sound argument leading to a contradiction.

Russell's Paradox

An alternative formulation of this problem is: "if the barber of Seville is a man who shaves all men in Seville who don't shave themselves, and only those men, who shaves the barber?"

Russell's Paradox

"Russell's response to the paradox is contained in his so-called theory of types. His basic idea is that we can avoid reference to S (the set of all sets that are not members of themselves) by arranging all sentences into a hierarchy. This hierarchy will consist of sentences (at the lowest level) about individuals, sentences (at the next lowest level) about sets of individuals, sentences (at the next lowest level) about sets of sets of individuals, etc. It is then possible to refer to all objects for which a given condition (or predicate) holds only if they are all at the same level or of the same "type"."

Src: http://plato.stanford.edu/entries/russell-paradox/

Logic Types

The HOL system's logic uses a simplification of Russell's type system due to Church with extensions developed by Milner [R7].

Types denote sets of values. Types can be atomic or compound.

Examples of atomic types are `bool`, `num`, `real`.

Compound types are built from other types using type operators.

Logic Types

Examples of compound types:

- $\sigma_1 \rightarrow \sigma_2$ is the set of functions with domain the type σ_1 and range the set σ_2
- $\sigma_1 \# \sigma_2$ is the set that is the cartesian product of the types σ_1 and σ_2

Higher order logic can have polymorphic functions.

The HOL system can infer the types of many expressions so we don't have to explicitly type every expression in the logic.

The ML function `type_of` can be useful to find out the types of logic terms.

See 3-hol-basics.sml

Axioms

HOL has very few primitive axioms. You can find out what they are by typing them at the ML prompt. For example:

```
BOOL_CASES_AX;
ETA_AX;
SELECT_AX;
INFINITY_AX;
```

These are ML identifiers associated with elements of type theorem.

See 3-hol-basics.sml

Primitive Rules of Inference

Src: Gordon [R5]

HOL has only 8 primitive rules of inference at its core. These are the only ways to create ML objects of type theorem. Everything else is derived from these. This is the secure core provided by the LCF implementation style.

We list the rules for completeness but omit the side conditions.

Assumption introduction: ASSUME

$$\frac{}{t \vdash t}$$

See 3-hol-basics.sml

Primitive Rules of Inference

Reflexivity: REFL

$$\frac{}{\vdash t = t}$$

Beta-conversion: BETA_CONV

$$\frac{}{\vdash (\lambda x.t_1)\ t_2 = t_1\ [t_2/x]}$$

Primitive Rules of Inference

Substitution: SUBST

$$\frac{\Gamma_1 \vdash t_1 = t_2 \qquad \Gamma_2 \vdash t[t_1]}{\Gamma_1 \cup \Gamma_2 \vdash t[t_2]}$$

Abstraction: ABS

$$\frac{\Gamma_1 \vdash t_1 = t_2}{\Gamma_1 \vdash (\lambda x.t_1) = (\lambda x.t_2)}$$

Primitive Rules of Inference

Type instantiation: INST_TYPE

$$\frac{\Gamma_1 \vdash t}{\Gamma_1 \vdash t[\sigma_1, \ldots, \sigma_n / \alpha_1, \ldots, \alpha_n]}$$

This is a substitution of types for type variables.

Discharging an assumption: DISCH

$$\frac{\Gamma \vdash t_2}{\Gamma - \{t_1\} \vdash t_1 \Rightarrow t_2}$$

Primitive Rules of Inference

Modus Ponens: MP

$$\frac{\Gamma_1 \vdash t_1 \Rightarrow t_2 \qquad \Gamma_2 \vdash t_1}{\Gamma_1 \cup \Gamma_2 \vdash t_2}$$

These are all ML functions!

See 3-hol-basics.sml

Soundness, Completeness, and Decidability

The HOL deductive system is sound (see p. 214 of [R4]).

The deductive system is not complete.

The logic is not decidable.

Outline

The HOL theorem proving system:

- Higher order logic
- Features of LCF-style theorem provers:
  - derived rules
  - theories
  - backward proof (tactics, tacticals)
- Examples
- HOL hammers
- Writing tactics
- Harry Potter
- DOIT_TAC

Milner's Three Great Ideas

1. Derived rules of inference so we can use "bigger" proof steps rather than always using the primitive steps
2. Theory as a record of facts already proven, and therefore available as lemmas (proofs can build on other proofs)
3. Tactics to organize the construction of backward proofs and tacticals to compose tactics (permits proof as a backward tree, rather than forward linear sequence). Milner's particular contribution was the method of translating the solution of the subgoals into the solution of the goals.

Milner's First Great Idea: Derived Rules

A derived rule combines the application of a sequence of rules.

In HOL, derived rules are correct by construction, i.e., they can't produce a non-theorem, because they are implemented in terms of the primitive inference rules.

Representing theorems as an abstract datatype and using the typing system of ML provides us with this guarantee.

A derived rule produces objects of type theorem.

From a user's perspective a derived rule is no different than a primitive rule.

Examples of Derived Rules

```
SYM      A |- t1 = t2
        --------------
         A |- t2 = t1
```

RAA:

```
CCONTR
           A, ~t |- F
        -----------------
            A |- t
```

Examples of Derived Rules

Exists-introduction: EXISTS

```
      A |- S[t]
-----------------
  A |- ?x. S[x]
```

Forall-elimination: SPEC

```
      A |- !x. S
------------------ SPEC `t`
  A |-  S[t/x]
```

See Mike Gordon's notes on the "References" course web page for the derived rules corresponding to the natural deduction proof rules we discussed last class.

See 3-hol-basics.sml

DECIDE

DECIDE is an example of powerful derived rule.

"The decision library contains co-operating decision procedures for quantifier-free formulas built up from linear natural number arithmetic, propositional logic, and the equational theories of pairs, recursive types, and uninterpreted function symbols. ... The procedure is based on the approach of Nelson and Oppen."

Src: HOL documentation

See 3-hol-basics.sml

Rewriting

Rewrite rules provide for limited "automatic" theorem proving. Rewriting involves pattern matching and substituting equals for equals.

HOL contains a number of derived rules based on rewriting that vary in whether the assumptions and built-in rewrite theorems are used.

```
- REWRITE_RULE;
> val it = fn : thm list -> thm -> thm
```

Rewriting can loop forever. For example, rewriting with ``!m n.  m + n = n + m``. Using `ONCE_REWRITE_RULE` can avoid this.

See 3-hol-basics.sml

Conversions

Conversions take terms and return theorems expressing the equality of that term to another term. `BETA_CONV` is an example of a conversion.

Conversionals are functions that combine conversions.

See 3-hol-basics.sml

Milner's Second Great Idea: Theories

In HOL, a theory is a set of theorems that have been proven. It's a means of storing groups of related information.

A theory can include: types, type operators, constants, definitions, axioms, and theorems.

A theory that does not introduce new axioms is called definitional.

When building a theory, we start the HOL session with `new_theory "mytheory";`.

Theories

For example, "numTheory" is a theory that defines the type of natural numbers, and the constants 0 and SUC. Peano's axioms have been proven in this theory. This theory also includes the induction theorem.

Another well-used theory is that of arithmetic. It includes definitions of plus, sub, mult, exponentiation, etc.

Many theories come with the HOL distribution. Look in the HOL distribution directory to see the theories available.

See 3-hol-basics.sml

Induction

Induction over numbers is a powerful proof technique. The theorem representing induction has been proven in the HOL system:

```
- numTheory.INDUCTION;
> val it =
    |- !P. P 0 /\
           (!n. P n ==> P (SUC n))
           ==> !n. P n : thm
```

Induction Proof Rule

Induction is so common a technique that a forward proof rule, and tactic (backward proof) exist for using the induction principle.

The derived forward proof rule INDUCT (found in `numLib`):

```
 A1 |- P[0]  A2 |- !n. P[n] ==> P[SUC n]
------------------------------------------ INDUCT
     A1 u A2 |- !n. P[n]
```

Theories

We can load existing theories using `load "theory-name";`

Opening a loaded theory makes all the names of the theory accessible without prefixing them with the theory name.

We can see what theories have been loaded using `print_theory "-".`

Once a theorem is proven, we can save it in the current theory using `save_thm ("thm-name",thm-ml-identifier)`

Building Theories

Most of the time in a verification effort, we'd like to build up our proofs incrementally.

We also might want to introduce new types and constants.

Definitions are conservative in HOL. This means by adding a definition we can't make the logic unsound.

Uninterpreted Types and Constants

An uninterpreted type is one for which we don't describe its composition. These are very useful for capturing a level of abstraction in the specification.

```
- new_type 0 "aircraft";
> val it = () : unit
```

An uninterpreted constant is one for which we don't supply a definition.

```
- new_constant
    ("flightLevel", ``:aircraft -> num``);
> val it = () : unit
```

For new types, if the arity is 0 (as above) it is a new base type.

Constant Definitions

```
val x_def = Define ‘ x = 1 ‘;

val divides_def =
  Define ‘divides a b = ?x. b = a * x‘;

val sum_def =
  Define ‘(sum 0 = 0) /\
          (sum (SUC n) = n + sum n)‘;
```

Definitions are theorems.

These constant definitions are not always executable.

See 3-hol-basics.sml

Total Functions

Higher order logic functions are total.

A total function is a function defined for all arguments of its domain.

A partial function is a function that is not defined for all arguments of its domain. Division is an example of a partial function. `HD` is also an example of a partial function.

In HOL, functions that are usually considered partial, such as `HD`, are total functions but they are only partially defined. We only define the behaviour of `HD` for certain arguments. The definition of `HD` is:

```
!h t. HD (h::t) = h
```

Recursive Functions

In classical logic, arbitrary recursive definitions aren't permitted.

Primitive recursive functions, which uniquely define functions are permitted.

HOL looks at the "form" of the definition to ensure that it is primitive recursive. For example, if it has the form of two cases, one for `0` and one for `SUC x` then it can tell it's okay to allow this definition.

The ML function "Define" tries to find a measure that shows the function will terminate and if it can, it does this proof behind the scenes.

See 3-hol-basics.sml

Type Introduction

We can introduce new types. Here's an enumerated type.

```
Hol_datatype
 `chocolate = Cadburys | Rogers | LauraSecord`;
```

The type definition package usually proves some useful theorems about your type. For example, in this case it proved the theorem:

```
chocolate_nchotomy
    |- !c. (c = Cadburys) \/ (c = Rogers)
           \/ (c = LauraSecord)
```

Use the following to store this theorem in an ML identifier, so it can be passed to inference rules and tactics:

```
val chocolate_thm = theorem "chocolate_nchotomy";
```

Recursive Types

Labelled binary trees:

```
Hol_datatype
  `bTree = Leaf of 'a
         | Node of 'bTree => 'bTree`;
```

The Hol_datatype function adds this type to the theory, and adds the constants `Leaf` and `Node` to the theory. It also proves a number of theorems about the type. For example:

```
bTree_distinct |-
  !a1 a0 a. ~(Leaf a = Node a0 a1)
```

Structural Induction

Another theorem proven about the type is:

```
bTree_induction
 |- !P.
      (!a. P (Leaf a)) /\
      (!b b0. P b /\ P b0 ==> P (Node b b0))
      ==> !b. P b
```

This second theorem allows us to do proof by structural induction on elements of this type.

Recursive Functions

One can define recursive functions on user-defined types:

```
val Leaves_def = Define
 `(Leaves (Leaf x) = 1) /\
  (Leaves (Node a b) = Leaves a + Leaves b)`;
```

`Define` uses some of the theorems previously proven about the type `bTree` to accept this as a recursive definition.

Lists

Lists are an example of a recursive type. They are found in the theory `listTheory`.

```
open listTheory;
```

Within this theory is the definition:

```
Hol_datatype `list = NIL | CONS of 'a => list`;
```

`NIL` and `CONS` are constructors for the elements of the type list.

This theory also defines operations such as `HD` and `TL`.

List Induction

There is also a structural induction principle for lists.

```
- list_induction;
> val it =
    |- !P. P [] /\
       (!t. P t ==> !h. P (h::t))
       ==> !l. P l : thm
```

In backwards proof, the tactic `Induct` will try to apply the relevant induction principle.

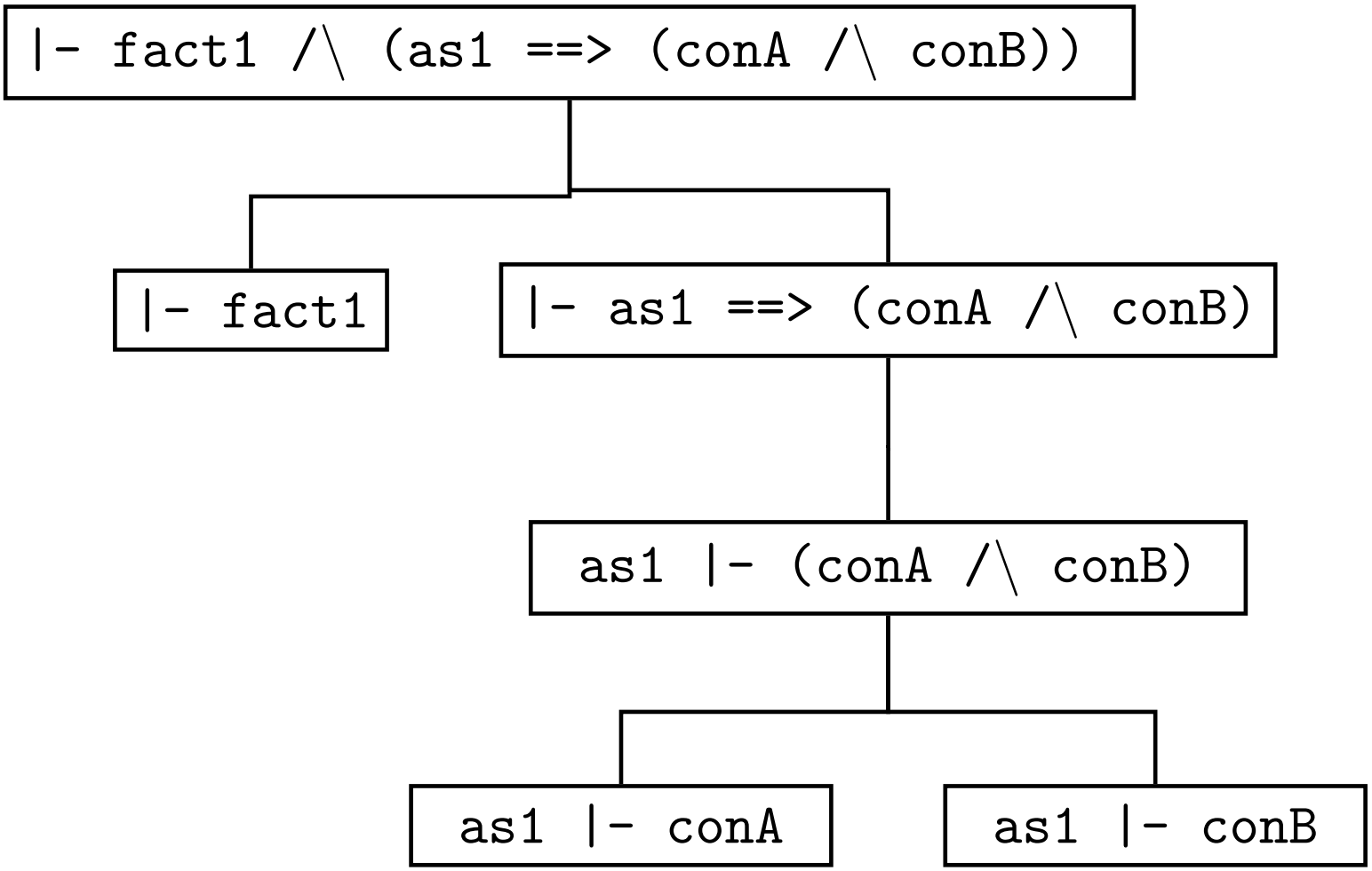
Milner's Third Great Idea: Tactics

Src: Gordon and Melham [R4], Ch. 24

In backward proof (goal-directed proof), we organize the search for the proof as a tree, starting with the objective, and decompose the goal successively. Each allowed decomposition comes with a way of translating the proofs of the subgoals into the proof of the goal. The decompositions can be thought of as proof strategies.

Example: To prove a goal that is a conjunction, decompose the goal into the two conjunct subgoals and in either order prove the two subgoals.

Example: Backward Proof



Example: Forward Proof

1	`{as1} |- conA`	premise
2	`{as1} |- conB`	premise
3	`{as1} |- conA /\ conB`	CONJ 1, 2
4	`|- as1 ==> conA /\ conB`	DISCH 3
5	`|- fact1`	premise
6	`|- fact1 /\ (as1 ==> conA /\ conB)`	CONJ 4,5

Tactics, Goals, and Justifications

A goal is a list of terms (assumptions), paired with a term (conclusion). These are the assumptions and conclusions of the theorem that we wish to prove.

```
goal = term list # term
```

A tactic is an ML function that when applied to a goal reduces it to a list of (sub)goals, along with a justification function mapping a list of theorems to a theorem. The justification function justifies the decomposition of the goal. The justification is essentially the forward proof step if we were doing a forward proof.

```
tactic = goal -> goal list # proof
proof = thm list -> thm
```

Tactics, Goals, and Justifications

As a user, you never have have to worry (or even see) the justification functions.

A tactic solves a goal if it reduces it to an empty set of subgoals. The simplest tactic that does this is one that recognizes a goal as an axiom or already proven theorem. This tactic is called ACCEPT_TAC.

Backword Proof (Goal Directed Proof)

Goal package consists of command that manipulate the proof stack:

- g ‘goal‘; sets the goal
- e(tactic); applies the tactic to the current goal
- backup(); backs up one step of the proof
- rotate x; moves around the list of subgoals
- drop(); drops the current proof being worked on
- p(); shows the current goal

See 3-hol-basics.sml

Tactics

Tactis are specified using a similar idea to natural deduction, except upside down.

```
        goal
-------------------------
 goal1  goal2  ...  goaln
```

These means the tactic reduces the goal above the line to the subgoals underneath the line.

Goals are written as `A ?- t`.

Common Tactics

CONJ_TAC

```
    A ?- t1 /\ t2
----------------------
A ?- t1         A ?- t2
```

Notice that the subgoals have the same assumptions as the original goal.

Theorem Tactics

Some tactics need an additional argument: they need a theorem to help them reduce the goal to a subgoal.

ACCEPT_TAC : thm_tactic

Solves a goal if the goal matches the supplied theorem (up to α-conversion).

Common Tactics

GEN_TAC

```
A ?- !x.t[x]
------------ GEN_TAC
A ?- t[x']
```

This is the reverse of forall-introduction. x' is arbitrary.

EXISTS_TAC

```
A ?- ?x.t[x]
------------  EXISTS_TAC `u`
A ?- t[u]
```

This is the reverse of exists-introduction.

Common Tactics

DISCH_TAC

```
    A ?- x ==> v
------------------- DISCH_TAC
  A u {x} ?- v
```

UNDISCH_TAC

```
    A u {x} ?- v
------------------- UNDISCH_TAC `x`
  A ?- x ==> v
```

Common Tactics

ASSUME_TAC

```
      A ?- t
------------------- ASSUME_TAC (A' |- x)
  A u {x} ?- t
```

where `A' |- x` is a previously proven theorem. A' must be a subset of A.

Common Tactics

ASM_CASES_TAC

```
            A ?- t
------------------------------- ASM_CASES_TAC `x`
A u {x} ?- t    A u {~x} ?- t
```

DISJ_CASES_TAC

```
   A ?- t
============== DISJ_CASES_TAC (A |- u \/ v)
A u {u} ?- t
A u {v} ?- t
```

Common Tactics

STRIP_TAC

This tactic removes universal quantifiers (GEN_TAC) (including those on an existential in the antecedent of an implication), puts antecedents of an implication in the assumption list (DISCH_TAC), and splits conjunctions into two subgoals (CONJ_TAC).

Sometimes STRIP_TAC does too much!

REWRITE_TAC, PURE_ONCE_REWRITE_TAC: backward versions of REWRITE_RULE.

SIMP_TAC: simplified the goal using the supplied "simpset" and additional theorems.
Example: SIMP_TAC std_ss [].

Induction Tactic

The tactic for using proof by induction splits the goal into two subgoals: a base case and a step case. `n` has type num in the following, and `n'` can't be free in `A`:

```
          A ?- !n. P
==================================== INDUCT_TAC
 A ?- P [0/n]   A u {P} ?- P[SUC n'/n]
```

See 3-hol-basics.sml

Commonly Used Tactics

IMP_RES_TAC: takes a theorem (an implication) and "resolves" the assumptions with the theorem to add more assumptions.

DECIDE_TAC: the backward version of DECIDE

PROVE_TAC: a big hammer! combines many of the available decision procedures coded as HOL derived rules including first order proof using the model elimination algorithm. PROVE_TAC takes a list of theorems as an argument.

Tacticals

Tacticals are ways of combining tactics. Tacticals can be used to create your own tactics so you can reuse proof strategies in multiple proofs.

Commonly used tacticals:

THEN: Sequencing

```
THEN : tactic -> tactic -> tactic
```

See 3-hol-basics.sml

Finding Theorems

In a system as large and powerful as the HOL system, it is unfortunately sometimes challenging to find the name of the relevant tactic to use, or to find the name of an existing theorem to re-use. Many people have already built useful theories.

You can use the ML function `match` to help find existing theorems:

```
- load "DB";   (* loaded and opened in startup.sml *)
- open DB;
- match [] ``x \/ ~x``;
> val it = [(("bool", "EXCLUDED_MIDDLE"),
             (|- !t. t \/ ~t, AncestorDB.Thm))]:
  ((string * string) * (thm * class)) list
```

Extensibility of the HOL System

The HOL system is extensible in two main ways:

1. programming new proof procedures: forward and backward (meta-language)
2. building re-usable theories (object language)

In building new theories, we can:

1. add new types (uninterpreted, enumerated, recursive)
2. add new constant definitions (possibly recursive)
3. add new uninterpreted constants

Using the above, we can embed other formalisms (such as temporal logic, statecharts, VHDL, etc.) and build proof support for these other notations.

Why higher order?

Why was important that the meta-language of the theorem prover be higher order?

Outline

The HOL theorem proving system:

- Higher order logic
- Features of LCF-style theorem provers:
  - derived rules
  - theories
  - backward proof (tactics, tacticals)
- Examples
- HOL hammers
- Writing tactics
- Harry Potter
- DOIT_TAC

What is Verification?

Recall from Week 1:

Verification involves checking a satisfaction relation, in the form of a sequent:

$$\mathcal{M} \models \phi$$

where

- $\mathcal{M}$ is a model (or implementation)
- ϕ is a property (or specification)
- $\models$ is a relationship that should hold between $\mathcal{M}$ and ϕ

Semantic Entailment

For propositional, first order, and higher order logic:

$$\alpha_1, \alpha_2, \alpha_3 \models \phi$$

means that in all Boolean valuations v where $v(\alpha_1) = \mathsf{T}$ and $v(\alpha_2) = \mathsf{T}$ and $v(\alpha_3) = \mathsf{T}$ then $v(\phi) = \mathsf{T}$, which is equivalent to saying

$$(\alpha_1 \wedge \alpha_2 \wedge \alpha_3) \Rightarrow \phi$$

is a tautology, i.e.,

$$\boxed{(\alpha_1, \alpha_2, \alpha_3 \models \phi) \quad \equiv \quad \models ((\alpha_1 \wedge \alpha_2 \wedge \alpha_3) \Rightarrow \phi)}$$

Verification using Higher Order Logic

We can write the description of a system in the logic. So $\mathcal{M}$ is a formula in logic.

We can also write the property ϕ as a formula in logic.

When we write the description of a system as a formula in the logic, a satisfying assignment for the formula represents a possible behaviour of the system.

The satisfaction relation then says all possible behaviours of the system satisfy the property ϕ.

Therefore, in this case, the satisfaction relation is implication.

Soundness

Soundness tells us that if using our theorem prover we show:

$$\vdash \mathcal{M} \Rightarrow \phi$$

then

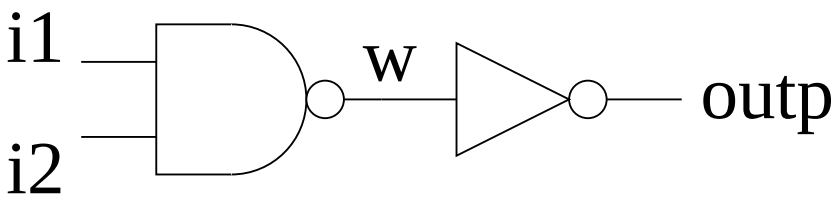
$$\models \mathcal{M} \Rightarrow \phi$$

and therefore from the previous slide:

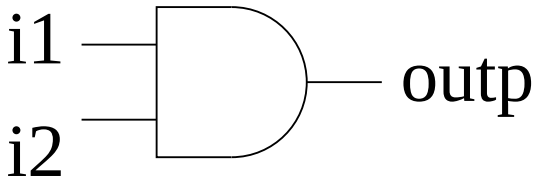
$$\mathcal{M} \models \phi$$

which is our satisfaction relation.

Example: And Gate



ANDGate_IMP



ANDGate

See 4-nand-and-ex.sml – three ways to do the same proof.

n-bit Ripple Carry Adder (5-n-adder-ex.sml)

For the n-bit ripple carry adder, we will have a model (implementation), and a property (specification) of its behaviour.

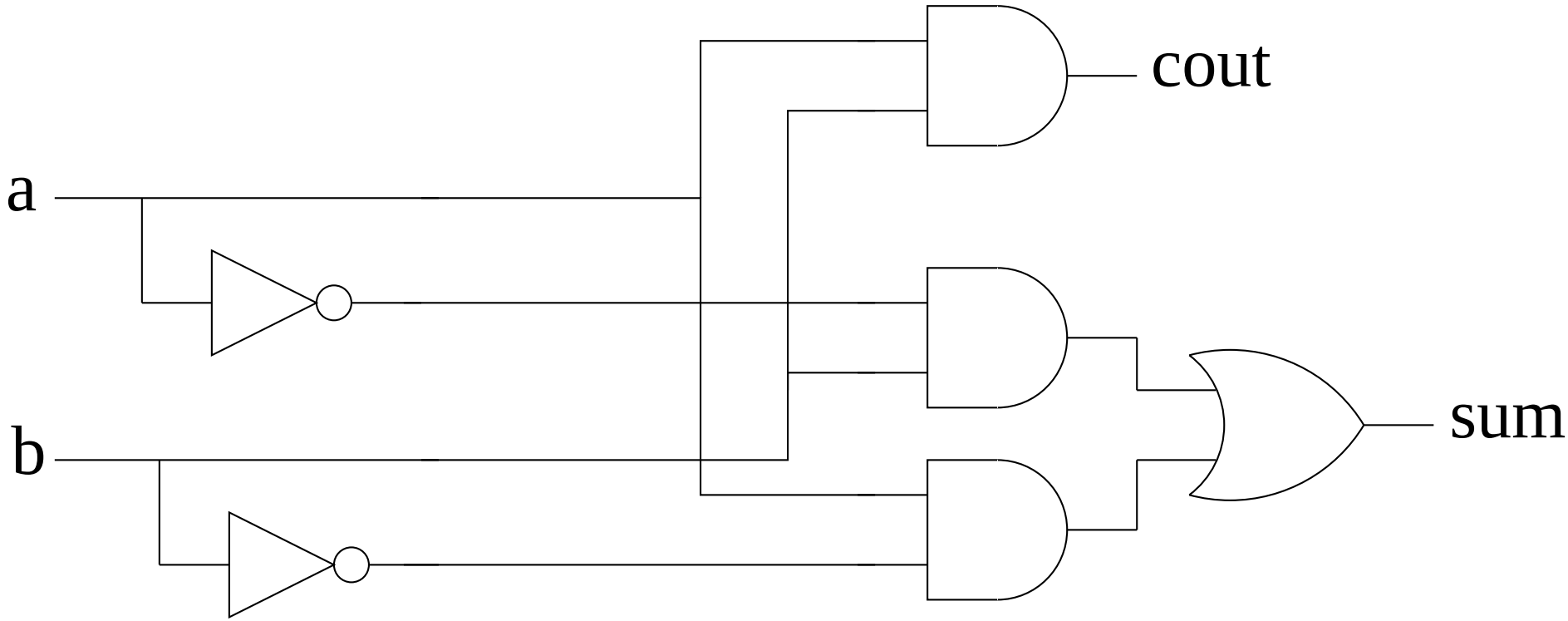
Both the model and the property will be expressed in higher order logic.

We will then prove using HOL that the implementation implies the specification. This is our satisfaction relation.

When both the model and property are expressed in the same logic, this means that any satisfying assignment of the implementation (behaviour) is also a satisfying assignment of the specification(behaviour).

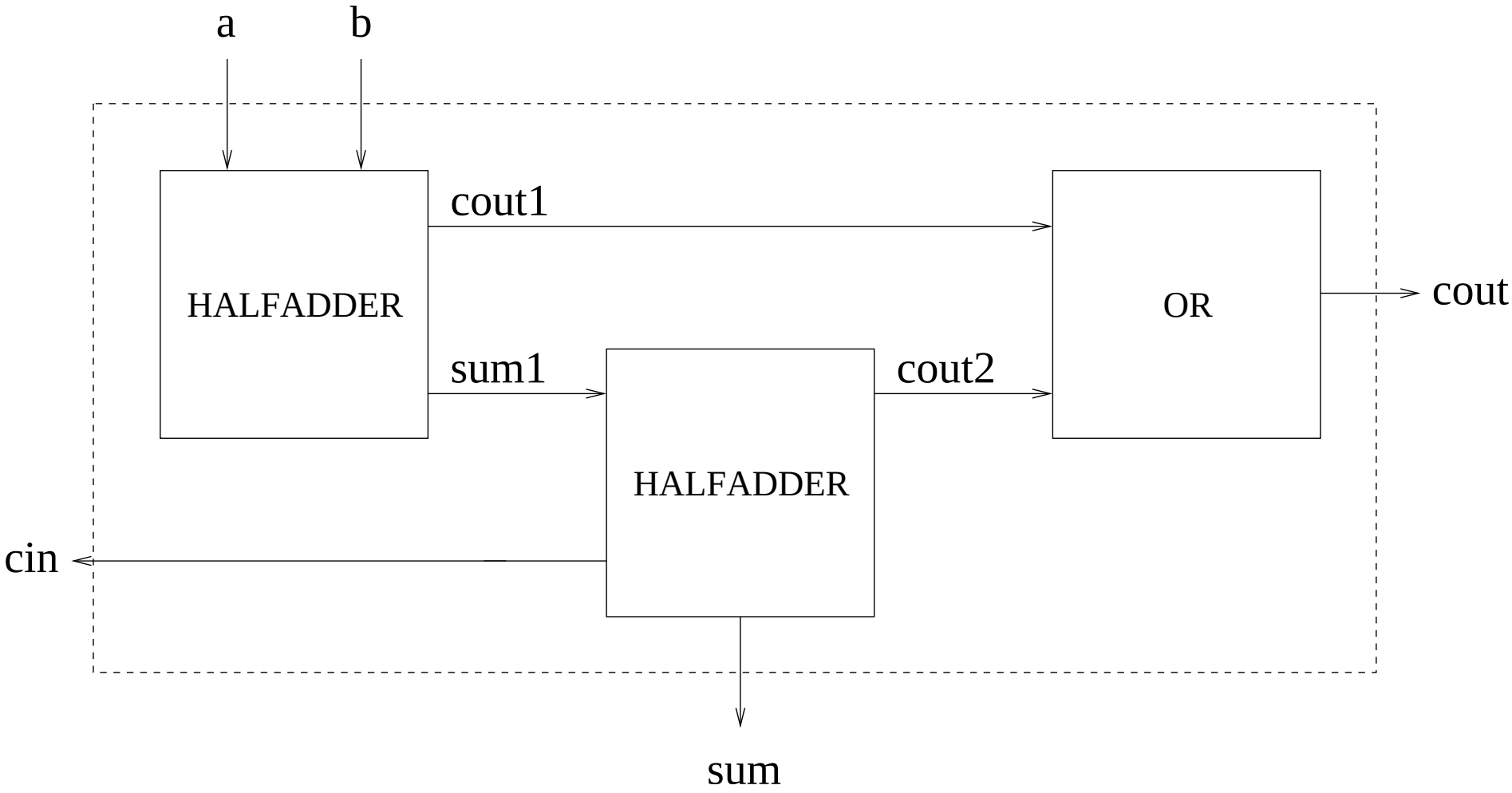
1-bit Half Adder

Add two bits.

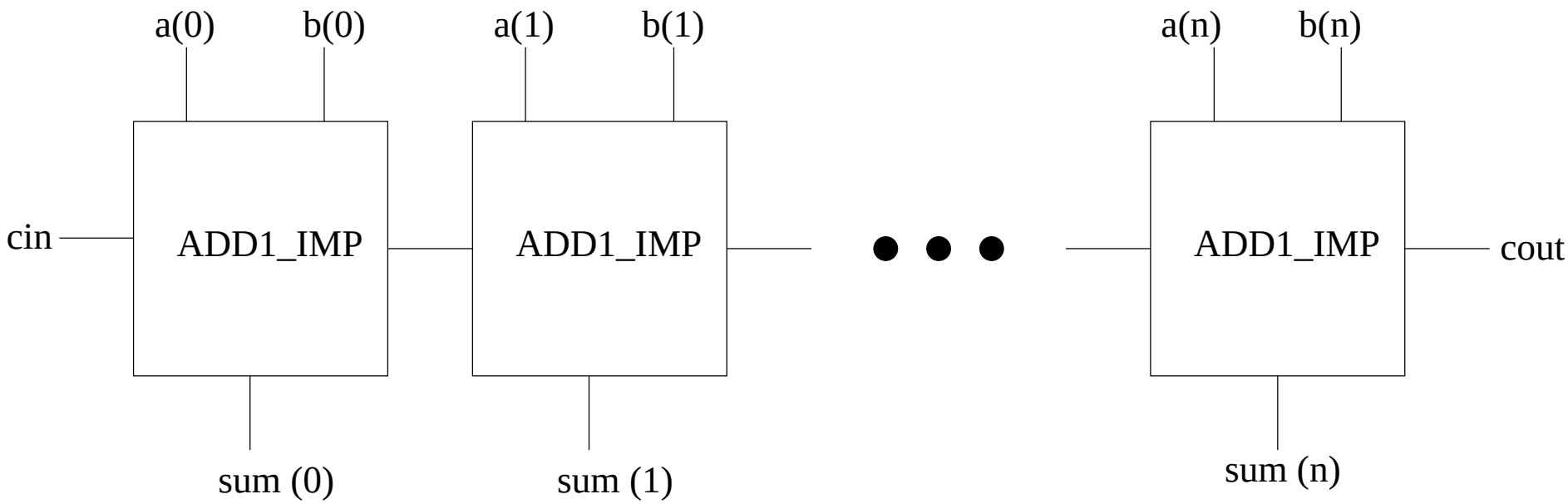


1-bit Full Adder (ADD1_IMP)

Add two bits and a previous carry.

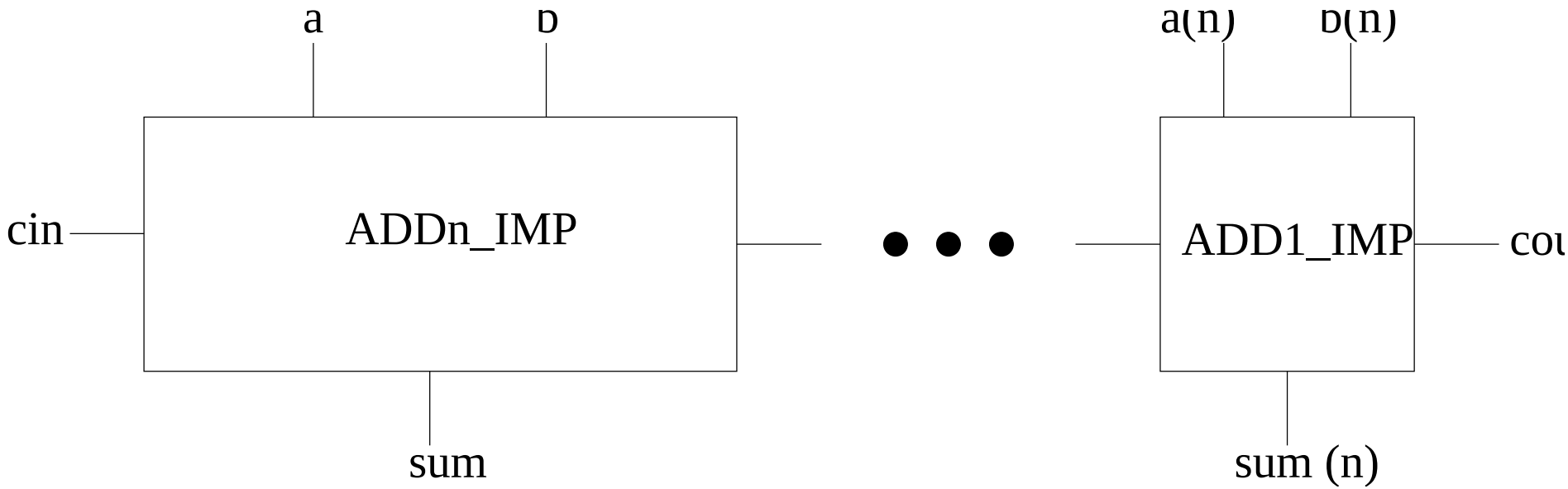


n-bit Ripple-Carry Adder



Bit 0 is the least significant bit.

n-bit Ripple-Carry Adder



Outline

The HOL theorem proving system:

- Higher order logic
- Features of LCF-style theorem provers:
  - derived rules
  - theories
  - backward proof (tactics, tacticals)
- Examples
- HOL hammers
- Writing tactics
- Harry Potter
- DOIT_TAC

HOL Hammers

- Working with assumptions
- Theories vs libraries
- Rewriting
- Automated reasoners
- Resolution
- Writing tactics

Working with Assumptions

Three options (possibly more):

1. referring to terms
2. matching a pattern
3. explicit numbering

All of these styles can be used in the same proof.

ASSUME_TAC allows us to add a theorem to the assumption list.

Quoted text in the following are from the on-line HOL reference guide.

Assumptions: referring to terms

Move assumptions to and from the antecedent of the goal.

- `e(DISCH_TAC)` "moves the antecedent of an implicative goal into the assumptions" (part of `STRIP_TAC`)
- `e(UNDISCH_TAC q)` "undischarges an assumption". Takes the assumption `q` (term frag list) off of the assumption list and places it as the antecedent of an implication in the goal.

Once in the goal, we can apply all the regular tactics to the assumption.

Assumptions: referring to terms

- `e(UNDISCH_THEN q ttac)` finds the first assumption equal to the term frag list `q` given, removes it from the assumption list, ASSUMEs it, passes it to the theorem-tactic (`ttac`) and then applies the consequent tactic.
- `e(DISCH_THEN ttac)` removes the antecedent and then creates a theorem by ASSUMEing it. This new theorem is passed to the theorem-tactic `ttac`. The consequent tactic is then applied.
  If the goal is `a ==> b`, `e(DISCH_THEN ttac)` grabs `a`, and creates the theorem `a |- a`, and then does `e(tac a)` on the goal `b` using the theorem tactical `ttac`.

Assumptions: matching a pattern

PAT_ASSUM (Tactical) "Finds the first assumption that matches the term frag list argument, applies the theorem tactic to it, and removes this assumption"

```
e(PAT_ASSUM `!x. P x` MATCH_MP_TAC);
```

Advantage: don't have to write the whole term.

Disadvantage: multiple assumptions may match the pattern.

Assumptions: explicit numbering

- `GA n` when assumption `n` is term `asm` returns the theorem $asm \models asm$
- `e(APP2ASM forward-rule n);` applies the forward-rule to assumption `n` and adds the result to the assumption list. It leaves assumption `n` on the list.
- `e(DROP [n1,n2,\dots]);` removes assumptions `n1` and `n2` from the assumption list.

Advantage: easier while creating a proof.

Disadvantage: harder when modifying a proof (assumption numbers change).

These functions are defined in `startup.sml` .

Theories

A theory contains theorems that have already been proven.

After starting hol:

```
- ancestry "scratch";
> val it =
    ["operator", "ind_type", "while", "num", "prim_rec", "r
     "arithmetic", "numeral", "normalForms", "one", "marker
     "bool", "sat", "sum", "option", "basicSize", "pred_set
  string list
```

shows the list of theories already loaded. To start using a theory other than these, use `load "theory-name"`.

Theories are structured as a hierarchy.

Theories

To see the theorems in a theory, `help "theory-name";`. (e.g., `help "numTheory"`)

You can access the theorems in a theory, using either

```
- numTheory.INDUCTION;
```

or

```
- open numTheory;

- INDUCTION;
```

Once opened a theory does not need to be re-opened.

Libraries

A library contains inference rules and tactics.

Both theories and libraries are grouped by topic (e.g., "numTheory" contains the induction theorem, "numLib" contains INDUCT_TAC. Note: the backward proof rule "Induct" is contained in bossLib.)

Accessing functions in a library is accomplished in the same way as accessing theorems in a theory.

The following libraries are pre-loaded when you run "hol.unquote": bossLib, simpLib, numLib, possibly more.

Rewrite Flavours

Recall that rewriting substitutes equals for equals.

1. REWRITE_TAC: takes a list of theorems and rewrites with these theorems and the system rewrite rules until no further rewriting can be done
2. PURE_REWRITE_TAC: as (1) but do not use the system rewrite rules
3. ONCE_REWRITE_TAC: as (1) but only make one rewriting pass. (This avoid infinite loops.)
4. PURE_ONCE_REWRITE_TAC: as (3) but without system rewrite rules

Rewrite Flavours

5. ASM_REWRITE_TAC: rewrite with the list of theorems and the assumed assumptions of the current goal
6. FILTER_ASM_REWRITE_TAC: as (5) but only use the assumptions which satisfy the filter predicate.

Also:
PURE_ONCE_ASM_REWRITE_TAC,
ONCE_ASM_REWRITE_TAC,
etc.

Automated Reasoners

These are all located in bossLib.

PROVE_TAC thmlist: first order reasoner; takes a list of theorems as an argument. Implementation based on technique of "model elimination".
"Some output (a row of dots) is currently generated as PROVE works. If the frequency of dot emission becomes slow, that is a sign that the term is not likely to be proved with the current lemmas." (You can then interrupt its execution.) If it fails, it doesn't change the goal.

Automated Reasoners

DECIDE_TAC: combination of decision procedure; handles statements of linear arithmetic and propositional logic.

Linear arithmetic: an equation in n variables $x_1, \ldots x_n$ of the form $c_1x_1 + c_2x_2 + \ldots c_nx_n = d$ where $c_1, c_2, \ldots c_n, d$ are constants. The variables are raised to the first power. Subtraction and the inequalities are also included.

Automated Reasoners

SIMP_TAC simp-set thmlist: simplifies a goal with the provided "simp-set" and theorems in "thmlist"
Common simp-sets are: std_ss, arith_ss, list_ss.
(These strictly increase in strength, meaning ones on the right contain ones on the left.)

RW_TAC simp-set thmlist: "Simplification with case-splitting and built-in knowledge of declared datatypes. . . . The case splits arising from conditionals and disjunctions can result in many unforeseen subgoals."

Related tactics are: STP_TAC, and ZAP_TAC (See p. 104 in the HOL description manual.)

Resolution by MP

IMP_RES_TAC thm: tries to repeatedly 'resolve' against the assumptions of a goal by attempting to match the antecedents the the theorem `thm` to the assumptions of the goal. Basically it tries to use MP on the provided `thm` and the assumptions of the goal.

RES_TAC: Similar to IMP_RES_TAC except that rather than working with a provided theorem, it tries to match the assumptions against each other using MP.

Resolution by MP

RES_TAC can take a long time and generate many additional assumptions that you don't need. (For an alternative, see the tactic that we will compose.)

These two tactics are "resolution by modus ponens", not the standard resolution for predicate logic that we will cover later in the course.

External Tools

HOL has an interface to well-known SAT solvers such as zchaff and grasp and a BDD packages.

See the libraries: HolSat, HolBdd.

Outline

The HOL theorem proving system:

- Higher order logic
- Features of LCF-style theorem provers:
  - derived rules
  - theories
  - backward proof (tactics, tacticals)
- Examples
- HOL hammers
- Writing tactics (6-new-tactic.sml)
- Harry Potter
- DOIT_TAC

Harry Potter Demo

- See handout for formalization and natural deduction examples.
- See 7-harry.sml for proof script.

Outline

The HOL theorem proving system:

- Higher order logic
- Features of LCF-style theorem provers:
  - derived rules
  - theories
  - backward proof (tactics, tacticals)
- Examples
- HOL hammers
- Writing tactics (6-new-tactic.sml)
- Harry Potter
- DOIT_TAC

When to use theorem proving

Interactive mechanized theorem proving can be both challenging and tedious. It takes a great deal of skill and patience, so you need to choose when to use a theorem prover carefully.

Theorem provers are good at:

1. Working at a higher level of abstraction and expressiveness
2. Deductive proof
3. Decomposition
4. Induction: numeric, structural, proofs of invariants
5. Complicated arithmetic
6. Embedding other notations

Example Applications of Theorem Proving

- Replicated hardware components
- Microarchitecture correctness
- Axiomatic software verification (Floyd-Hoare Logic)
- Algorithm verification
- Abstract requirements validation

Summary

- Mechanized theorem provers do pattern-matching well.
- Capabilities:
  - Bookkeeping
  - Proof Checking
  - Automated Proof Procedures
- Higher order logic: quantification over functions; functions can take other functions are arguments
- LCF-style theorem proving:
  - Separation of object language and meta-language
  - Meta-language is a typed functional language (ML) – secure core
  - Backward proof (tactics, tacticals)
  - Extensibility:
    - Derived rules/tactics
    - Theories (theorems, conservative extension by definition: types, constants,)

References

R1. R. L. Constable et al. Implementing Mathematics with The Nuprl Proof Development System. Prentice Hall, 1995.

R2. Alonzo Church. A formulation of the simple theory of types. Journal of Symbolic Logic, 5(2):5668, June 1940.

R3. W. M. Farmer. A partial functions version of Churchs simple theory of types. Journal of Symbolic Logic, 55:12691291, 1990.

R4. M.J.C. Gordon and T.F. Melham, editors. Introduction to HOL. Cambridge University Press, 1993.

R5. M. Gordon. Hol: A machine oriented formulation of higher order logic. Technical Report 68, Computer Laboratory, University of Cambridge, July 1985.

References

R6. Gerard Huet, Gilles Kahn, and Christine Paulin-Mohring. The coq proof assistant a tutorial version 7.0, 2001. http://pauillac.inria.fr/coq/doc/tutorial.html.

R7. R. Milner. A theory of type polymorphism in programming. Journal of Computer and System Sciences, 17(3):348375, 1978.

R8. NASA. Formal methods specification and analysis guidebook for the verification of software and computer systems volume ii: A practitioners companion. http://eis.jpl.nasa.gov/quality/Formal Methods/.

R9. S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In Conference on Automated Deduction, volume 607 of Lecture Notes In Computer Science, pages 748752, 1992.

R10. Lawrence C. Paulson. Intoduction to Isabelle, 2001. Isabelle Distribution.