HOL, Part 2

More involved manipulation of goals

- Imagine *A,B* ?-   *hyp*

- I want to :
  - Rewrite *hyp* using *A*   // ok
  - I know *A* implies *A'* ; I want to use *A'* to reduce *hyp*
  - Rewrite *B*

- I only want to rewrite some part of the hypothesis

Theorem Continuation

(Old Desc 10.5)

- Is an (ML) function of the form:

$$\boldsymbol{tc} \;:\; (thm \rightarrow tactic) \rightarrow tactic$$

tc f typically takes one of the goal's assumptions (e.g. the first in the list), ASSUMEs it to a theorem *t*, and gives *t* to *f*. The latter inspects *t*, and uses the knowledge to produce a new tactic, which is then applied to the original goal.

- Useful when we need a finer control on using or transforming *specific* assumptions of the goal.

Example

Goal: $\quad$ *assumptions* $\;$ *?-* $\;$ *ok 10*

Contain $\;$ "$(\forall n.\ P\ n \Rightarrow ok\ n)$"

assumptions $\;$ *?-* $\;$ *P 10*

So, by MP we should be able to reduce to the one on the right:

But how?? With the tactic below:

$MATCH\_MP\_TAC\ :\ thm \rightarrow tactic$

FIRST_ASSUM $\quad$ *MATCH_MP_TAC*

" *assumptions* $\;$ *?-* $\;$ *ok 10*"

$FIRST\_ASSUM\ :\ (thm \rightarrow tactic) \rightarrow tactic$

Some other theorem continuations

- $POP\_ASSUM : (thm \rightarrow tactic) \rightarrow tactic$
- $ASSUM\_LIST : (thm\ list \rightarrow tactic) \rightarrow tactic$
- $EVERY\_ASSUM : (thm \rightarrow tactic) \rightarrow tactic$
- etc

Variations

- In general, exploiting higher order functions allows flexible programming of tactics. Another example:

$$RULE\_ASSUM\_TAC : \quad (thm \rightarrow thm) \rightarrow \ tactic$$

RULE_ASSUM f maps f on all assumptions of the target goal; it fails if f fails on one asm.

- Example:

```
RULE_ASSUM_TAC   (fn   thm =>    SYM  thm   handle _ => thm)
```

Conversion

(Old Desc Ch 9)

- Is a function to generate *equality* theorem → $\boxed{\vdash\ t=u}$
- Type: $\boxed{conv\ =\ term \rightarrow thm}$ such that if *c:conv*

  then *c t* can produce $\boxed{\vdash\ t = something}$

- We have seen one: BETA_CONV; but HOL has *lots* of conversions in its library.
- Used e.g. in rewrites, in particular rewrites on a specific part of the goal.

Examples

- BETA_CONV “(\x. x) 0” → |- (\x. x) 0 = 0

- COOPER_CONV “1>0 → |- 1>0 = T

- FUN_EQ_CONV “f=g” →

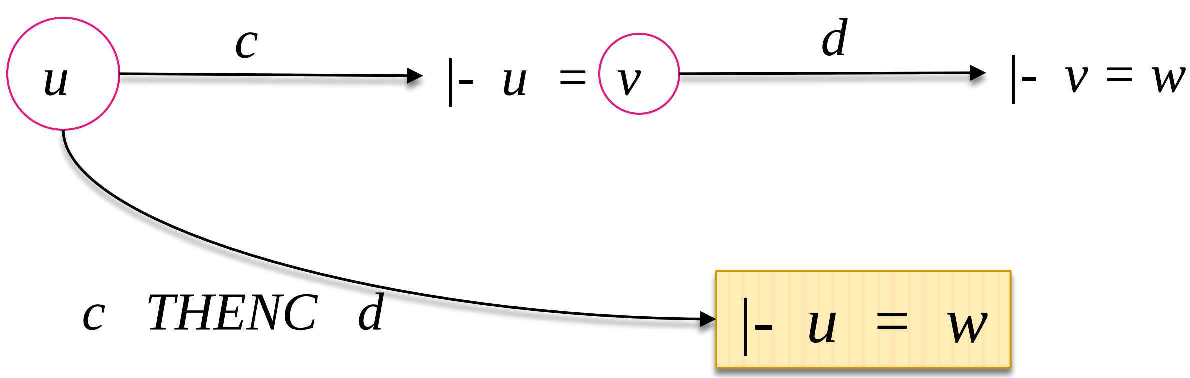
|- (f=g) = (!x. f x = g x)

Composing conversions

- The unit and zero: ALL_CONV, NO_CONV

- Sequencing: *c THENC d*

If *c* produces |- *u*=*v*, *d* will take *v*; if *d v* then produces |- *v*=*w*, the whole conversion will produce |- *u*=*w*.



Composing conversions

- Try c; but if it fails then use d.

$$c \quad ORELSEC \quad d$$

- Repeatedly apply c until it fails:

$$REPEATC \quad c$$

And tree walking combinators ...

- Allows conversion to be applied to specific subtrees instead of the whole tree:

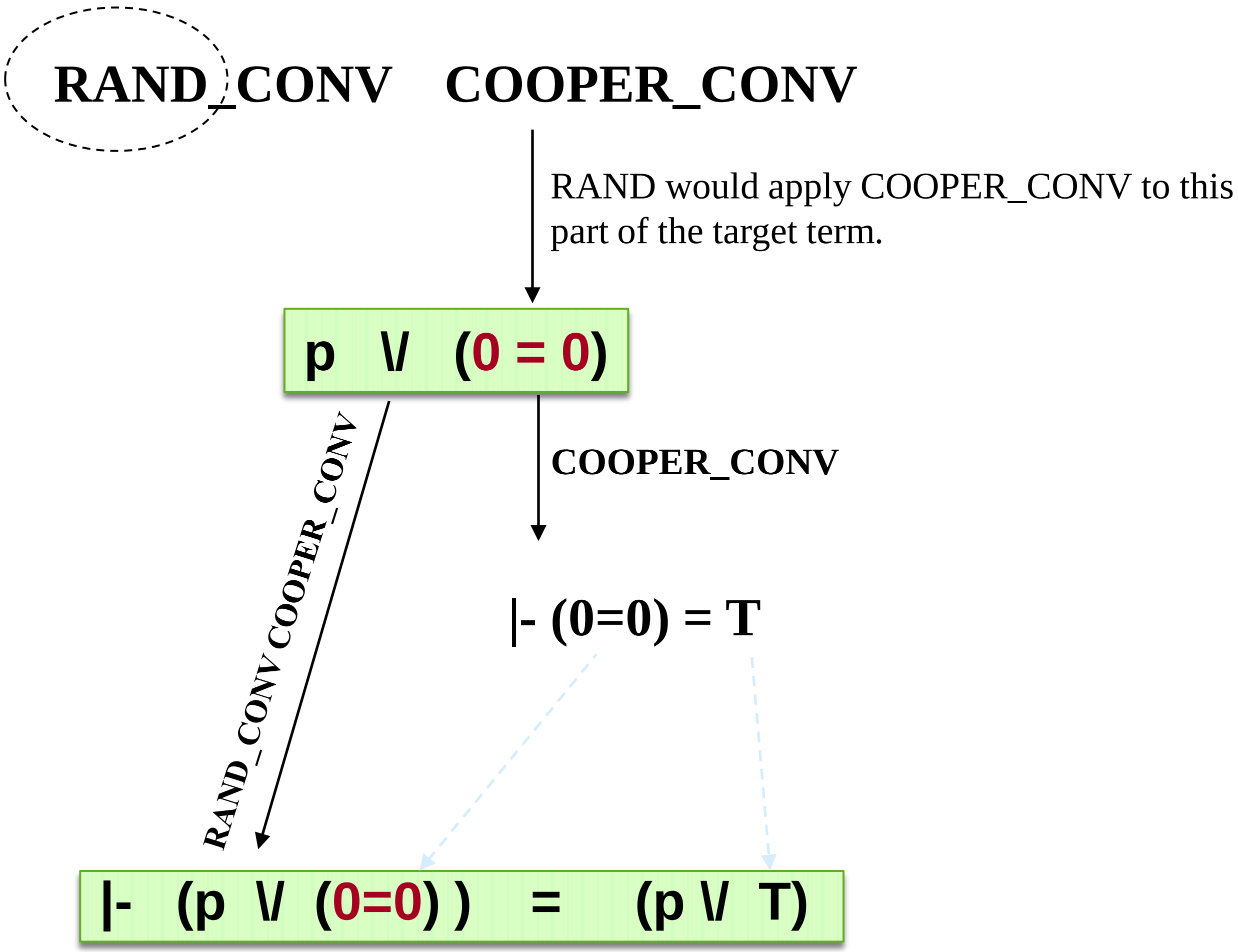
$$RAND\_CONV : conv \rightarrow conv$$

$RAND\_CONV\ c\ t$ applies c to the 'operand' side of t.

- Similarly we also have RATOR_CONV → apply c to the 'operator' side of t

- You can get to any part of a term by combining these kind of combinators.

Example



RAND_CONV COOPER_CONV
RAND would apply COOPER_CONV to this part of the target term.
p \/ (0 = 0)
COOPER_CONV
RAND_CONV COOPER_CONV
|- (0=0) = T
|- (p \/ (0=0)) = (p \/ T)

Tree walking combinators

- We also have combinators that operates a bit like in strategic programming ☺

- Example: $$DEPTH\_CONV \ : \ conv \rightarrow \ conv$$

DEPTH_CONV c t will walk the tree t (bottom up, once, left to right) and repeatedly applies c on each node.

- Variant: ONCE_DEPTH_CONV

- Not enough? Write your own?

Examples

- DEPTH_CONV **BETA_CONV** t

→ would do BETA-reduction on every node of t

- DEPTH_CONV **COOPER_CONV** t

→ use COOPER to simplify every arithmetics subexpression of t

e.g. 1>0 /\ p → |- 1>0 /\ p = T /\ p

Though in this case it actually does not terminate because COOPER_CONV on “T” produces “|- T=T”

Can be solved with CHANGED_CONV.

Turning a conversion to a tactic

- You can lift a conv to a rule or a tactic ☺

$$CONV\_RULE : conv \rightarrow rule$$

$$CONV\_TAC \quad : conv \rightarrow tactic$$

- *CONV_TAC c "A ? t"*

would apply c on t; suppose this produces |- $t{=}u$, this this theorem will be used to rewrite the goal to A ? u.

- Example: ?- $\sim (f = g)$

To expand the inner functional equality to point-wise equality do:

CONV_TAC (RAND_CONV FUN_EQ_CONV)

Primitive HOL

Implementing HOL

- An obvious way would be to start with an implementation of the predicate logic, e.g. along this line:

```
data  Term  =  VAR  String
            |  OR    Term  Term
            |  NOT   Term
            |  EXISTS  String  Term
            |  ...
```

- But want/need more:
  - We want terms to be typed.
  - We want to have more operators
  - We want to have functions.

Building ontop (typed) λ- calculus

- It's a clean and minimalistic formal system.
- It comes with a very natural and simple type system.
- Because of its simplicity, you can trust it.
- Straight forward to implement.
- You can express functions and higher order functions very naturally.
- We'll build our predicate logic ontop of it; so we get all the benefit of λ-calculus for free.

λ- calculus

- Grammar:

term ::=	*var*		
	|	*const*	
	|	*term term*	// e.g. $(\lambda x.\ x)\ 0$
	|	*\var. term*	// e.g. $(\lambda x.\ x)$

- The terms are typed; allowed types:

type ::=	*tyvar*		// e.g. ‘a
	|	*tyconst*	// e.g. bool
	|	*(type,…,type) tyop*	// e.g. bool list
	|	$type \to type$	

λ- calculus computation rule

- One single rule called β-reduction

$$(\lambda x.\, t)\ u \quad \to \quad t[u/x]$$

- However in theorem proving we're more interested in concluding whether two terms are 'equivalent', e.g. that:

$$(\lambda x.\, t)\ u \ = \ t[u/x]$$

- So we add the type "bool" and the constant "=" of type:

$$'a \to\ 'a \to bool$$

HOL Primitive logic

(Desc 1.7)

- These inference rules are then the minimum you need to add (implemented as ML functions):

ASSUME (t:bool) = [t] |- t

REFL t = |- t=t

BETA_CONV "(\x. t) u"

=

|- (\x. t) u = t[u/x]

HOL Primitive logic

$ABS\ \text{“}\vdash t=u\text{”} \quad = \quad \vdash\ (\backslash x.\ t) = (\backslash x.\ u)$

$SUBST\ \text{“}\vdash x=u\text{”}\ \ t \quad = \quad \vdash\ t = t[u/x]$

$INST\_TYPE\ (\alpha,\tau)\ \text{“}\vdash t\text{”} \quad = \quad \vdash\ t[\tau/\alpha]$

HOL Primitive logic

In λ-calculus you also have the η-conversion that says:

$$f = g \quad \text{iff} \quad (\forall x.\ f\ x = g\ x)$$

This is formalized indirectly by, later, this axiom:

ETA_AX: $\vdash \forall f.\ \ (\lambda x.\ f\ x) \ = \ f$

HOL Primitive logic

- We'll also add the constant "$\Rightarrow$", whose logical properties are captured by the following rules:

$$DISCH \quad \text{“}t, A \;|\!-\; u\text{”} \quad = \quad A \;|\!-\; t \Rightarrow u$$

$$MP \;\; thm_1 \; thm_2 \quad \rightarrow \; \textit{implementing the modus ponens rule}$$

Predicate logic
(Desc 3.2)

- So far the logic is just a logic about equalities of λ-calculus terms.

- Next we want to add predicate logic, but preferably we build it in terms of λ-calculus, rather than implementing it as a hard-wired extension to the λ-calculus.

- Let's start by declaring two constants T,F of type bool with the obvious intent. Now find a way to encode the intent of "T" in λ-calculus $\rightarrow$ captured by this definition:

$$T\_DEF: \quad \vdash \quad T \quad = \quad ((\lambda x{:}bool.\ x) \ = \ (\lambda x.\ x))$$

Encoding Predicate Logic

(Desc 3.2)

- Introduce constant "$\forall$ "of type ('a$\rightarrow$bool)$\rightarrow$bool, defined as follows:

$$FORALL\_DEF : \quad |\!- \quad \forall P \quad = \quad (P = (\lambda x.\ T\ ))$$

which HOL pretty prints as $(\forall x.\ P\ x)$

- Now we define "F" as follows:

$$F\_DEF : \quad |\!- \ F \quad = \quad \forall t{:}bool.\ t$$

- Puzzle for you: prove just using HOL primitive rules (more later) that $\neg$(T = F).

Encoding Predicate Logic

- *NOT_DEF :* $\vdash\ \forall p.\ \ {\sim}p \ = \ p \Rightarrow F$
- *AND_DEF:* $\vdash\ \forall p\, q.\ \ p \wedge q \ = \ {\sim}(p \Rightarrow {\sim}q)$
- *OR_DEF ...*
- *SELECT_AX:* $\vdash\ \forall P\, x.\ \ P\, x \ \Rightarrow\ P\ (@P)$
- *EXISTS_DEF :* $\vdash\ (\exists x.\ P) \ = \ P\ @P$

And some axioms ...

- *BOOL_CASES_AX:* $\vdash \forall b.\ (b=T) \vee (b=F)$
- *IMP_ANTISYM:*

$$\vdash \forall b_1\ b_2.\ (b_1 \Rightarrow b_2) \Rightarrow (b_2 \Rightarrow b_1) \Rightarrow (b_1=b_2)$$

And this infinity axiom...

We declare a type called “ind”, and impose this axiom:

INFINITY_AX :

$$\vdash \exists f{:}ind \to ind.\ \ One\_One\ f \ \ /\backslash \ \ \sim Onto\ f$$

This indirect says that there “ind” is a type with infinitely many elements!

One One f = $\forall$x y. (f x = f y) $\Rightarrow$ (x = y) // every point in rng f has at most 1 source
Onto f = $\forall$y. $\exists$x. y = f x . // every point in rng f has at least 1 source
// also keep in mind that all function sin HOL are total

Examples of building a derived rules

$UNDISCH\ \text{“}A\ |\!-\ t \Rightarrow u\text{”} \quad = \quad t,A\ \ |\!-\ \ u$

fun UNDISCH thm_1 = // $A\ |\!-\ t \Rightarrow u$

let

thm_2 = ASSUME t // $t\ |\!-\ t$

thm_3 = MP thm_1 thm_2 // $t,A\ \ |\!-\ \ u$

in thm_3 end

Note: this is just a pseudo code; not a real ML code.

Examples of building a derived rules

SYM “A |- $t = u$” = A |- $u = t$

fun SYM thm_1 = // A |- $t = u$

let

thm_2 = REFL t // |- $t = t$

thm_3 = SUBST { “x” $\rightarrow$ thm_1 } “x=c” thm_2 // A |- $u{=}t$

in thm_3 end

Proving ~(T = F)

thm_1 = REFL "(λx. x)" // $\vdash (\lambda x.\ x) = (\lambda x.\ x)$

TRUTH = SUBST ... (SYM T_DEF) thm_1 // $\vdash T$

thm_2 = ASSUME "T=F" // $T{=}F \ \vdash\ T{=}F$

thm_3 = SUBST ... thm_2 TRUTH // $T{=}F \ \vdash\ F$

thm_4 = DISCH "T=F" thm_3 // $\vdash (T{=}F) \Rightarrow F$

thm_5 = SUBST ... (SYM … NOT_DEF) thm_4 // $\vdash\ \sim(T{=}F)$

extending HOL with new types

Extending HOL with your own types

- The easiest way to do it is by using the ML function HOL_datatype, e.g. :

Hol_datatype *`RGB = RED | GREEN | BLUE`*

Hol_datatype *`MyBinTree = Leaf int | Node MyBinTree MyBinTree*

which will make the new type for you, and *magically* also conjure a bunch of ‘axioms’ about this new type ☺.

- We’ll take a closer look at the machinery behind this.

Defining your own type, from scratch.

- To do it from scratch we do:

```
new_type  ("RGB",0) ;
```

- and then declare these constants:

```
new_constant  ("RED", Type `:RGB`) ;
new_constant  ("GREEN", Type `:RGB`) ;
new_constant  ("BLUE", Type `:RGB`) ;
```

- Is this ok now ?

To make it exactly as you expected, you will need to impose some axionms on RGB…

```
new_axiom("Axiom1",
--`
   ~(RED= GREEN) /\ ~(RED = BLUE) ...
`--) ;
```

$(\forall c{:}RGB.\ (c=RED) \lor (c=GREEN) \lor (c = BLUE))$

(basically, we need to make sure that RGB is isomorphic to {RED,GREEN,BLUE})

Defining a recursive type, e.g. "num"

- We declare a new type "num", and declare its constructors:
  - $0 : num$
  - $SUC : num \rightarrow num$
- Add sufficient axioms, we'll use Peano's axiomatization:

$$(\forall n.\ \ 0 \neq SUC\ n\ )$$

$$(\forall n.\ \ (n=0)\ \lor\ (\exists k.\ n = SUC\ k))$$

$$(\forall P.\ \ P\ 0\ \ \land\ \ (\forall n.\ P\ n \Rightarrow P\ (SUC\ n)\ )\ \Rightarrow\ (\forall n.\ P\ n)\ )$$

Defining “num”

- And this axiom too:

$$(\forall e\ \oplus.\quad (\exists f.\ \ (f\ 0 = e)\ \ \wedge\ \ (\forall n.\ f\,(SUC\ n) = n \oplus (f\ n))\ )$$

- which implies that equations like:

$$\boxed{\begin{array}{l} sum\ 0 = 0 \\ sum\ (SUC\ n)\ =\ n + (sum\ n) \end{array}}$$

define a function with exactly the above properties.

But ...

- Just adding axioms can be dangerous. If they're inconsistent (contradicting) the whole HOL logic will break down.

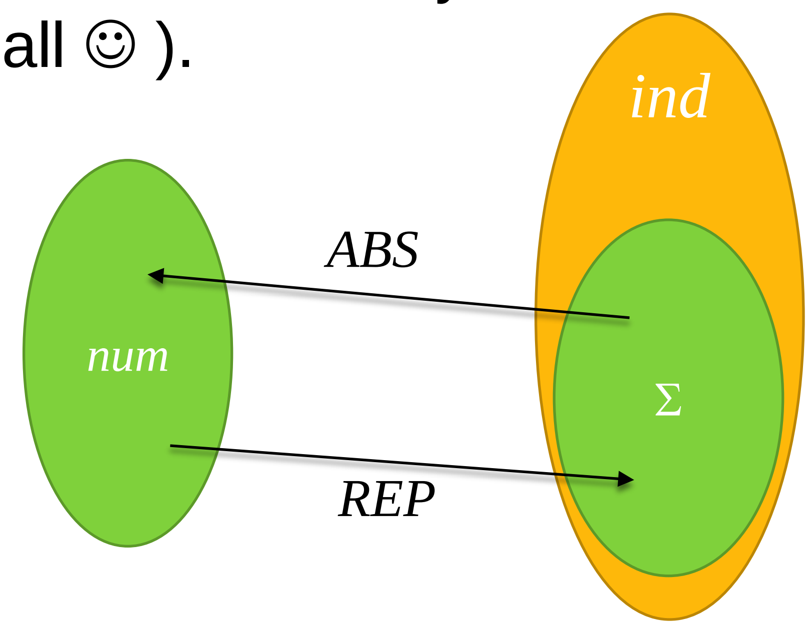
- Contradicting type axioms imply that your type τ is actually empty. So, e.g. β-reduction should <u>not</u> be possible:

$$\vdash \ (\lambda x{:}\tau.\ \ P)\ \ e \ \ = \ \ P[e/x]$$

However HOL requires types to be non-empty; its β-reduction will always succeed.

Definitional extension

- A safer way is to define a ‘bijection’ between your new type and an existing type.
- At the moment the only candidate is “ind” (“bool” would be too small ☺).



- Now try to prove the type axioms from this bijection → safer!

First characterize the Σ part…

- First, define REP_{SUC} as the function f:ind→ind that INFINITY_AX says to exist. That is, f satisfies:

$$ONE\_ONE\ f \ \ \wedge \ \ {\sim}ONTO\ f$$

- "REP_{SUC}" is the model of "SUC" at the ind-side.
- Similarly, define REP_0 as the model of 0:

$$REP_0 = @(\lambda z{:}ind.\ {\sim}(\exists x.\ z = SUC_{REP}\ x))$$

So, REP_0 is some member of "ind" who has no f-source (or SUC_{REP} source).

The Σ part

- Define Σ as a subset of ind that admits num-induction.

We'll encode Σ as a predicate ind$\rightarrow$bool:

$$\Sigma x \quad = \quad (\forall P.\ \ P\ \ REP_0\ \wedge\ (\forall y.\ P\ y \Rightarrow P\ (SUC_{REP}\ y)) \Rightarrow\ \ P\ x)$$

So, *x*:ind represents a num, iff:

for any P satisfying num-induction's premises, P holds on x.

Defining "num"

- Now postulate that num can be obtained from Σ by a the following bijection. First declare these constants:

$$\boxed{\begin{array}{l} rep : num \to ind \\ abs : ind \to num \end{array}}$$

- Then add these axioms:

$\boxed{rep \text{ is injective}}$ $\boxed{(\forall n{:}num.\ \Sigma(rep\ n))}$

$\boxed{(\forall n{:}num.\ abs(rep\ n) = n)}$ $\boxed{(\forall x{:}ind.\ \Sigma x \Rightarrow rep(abs\ x) = x)}$

$$\boxed{\begin{array}{lcl} rep\ 0 & = & REP_0 \\ rep\ (SUC\ n) & = & REP_{SUC}\ (rep\ n) \end{array}}$$

Now you can actually prove the orginინal axioms of num

- E.g. to prove 0 ≠ SUC n; we prove this with contradiction:

$$
\begin{array}{ll}
 & 0 = SUC\ n \\
\Rightarrow & \\
 & rep\ 0 \ = \ rep\ (SUC\ n) \\
= & \text{// with axioms defining reps of 0 and SUC} \\
 & REP_0 \ = \ REP_{SUC}\ (rep\ n) \\
\Rightarrow & \text{// def. } REP_0 \\
 & F
\end{array}
$$

Automated

- Fortunately all these steps are automated when you make a new type using the function Hol_datatype. E.g. :

Hol_datatype `NaturalNumber = ZERO | NEXT of NaturalNumber

will generate the 4 axioms you saw before. e.g :

NaturalNumber_distinct : $\vdash \; \forall n.\ \sim(ZERO = NEXT\ n)$

NaturalNumber_induction :

$\vdash \; \forall P.\ \ P\ ZERO \wedge (\forall n.\ P\ n \Rightarrow P\ (NEXT\ n)) \Rightarrow (\forall n.\ P\ n))]$