HOL, Part 2

Automating my own logic

UNITY

- Based on UNITY, proposed by Chandy and Misra, 1988, in *Parallel Program Design: a Foundation*. Later, 2001, becomes Seuss, with a bit OO-flavour in: *A Dicipline of Multiprogramming: Programming Theory for Distributed Applications*

- Unlike LTL, UNITY defines its logic Axiomatically:
  - more abstract
  - suitable for deductive style of proving
  - you can deal with infinite state space
  - but it is not a counter example based logic

Example

init : empty(s) $\wedge$ x=0

actions:

$\neg$empty(s) $\rightarrow$ y := retrieve(s)

[]

x++

[]

isprime(x) $\rightarrow$ add(x,s)

No control structure → we focus on concurrency, and we try to model that abstractly, by simply specifying *when* the activities can be scheduled, rather than how they are scheduled.

UNITY Program & Execution

- A program P is (simplified) a pair *(Init,A)*

  Init : a predicate specifying allowed initial states
  A : a set of concurrent (atomic and guarded) actions

- Execution model :

  - Each action α is executed atomically. Only when its guard is enabled (true), α can be selected for execution.

  - A run of P is *infinite*. At each step an enabled action is *non-deterministically* selected for execution. The run has to be *weakly fair:* when an action is persistently enabled, it will eventually be selected. When no action is enabled, the system stutters (does a skip).

However the logic is axiomatic. It will not actually construct the runs. → next slides.

Temporal properties

- Safety is expressed by this operator:

$$(Init, A) \ \vdash \ p \ \textbf{unless} \ q \ = \ \forall \alpha \in A. \ \{ p \wedge \neg q \} \ \alpha \ \{ p \vee q \}$$

Whenever p holds the program will either stay in p, or go over to q.

- This roughly corresponds to $\Box(p \rightarrow p \ \mathbf{W} \ q)$
- An LTL property is quantified over executions of the program *P* (not over all executions!). This is problematical for constructing its proof.
- A UNITY property totally ignores executions.

Example

init : pc=0 /\ x=0
actions:
  pc=0 → x,pc := x+1, 1
[] pc=1 → x,pc := x+1, 2
[] pc=2 → skip

{ x+1 ;
 x+1 ;
 repeat skip }

- This is valid: □(x=2 → x=2 **W** false)
- But this is not valid: x=2 **unless** false , however this is: pc∈{0,1,2} /\ (pc=0 ⇔ x=0) /\ (pc=1 ⇔ x=1) /\ (pc=2 ⇔ x=2) /\ x=2 **unless** false
- Note that a direct LTL proof will also have to somehow construct those intermediate information.

Temporal properties

- A predicate *p* is transient in *P=(I,A)* if there is an action in *A* that can make it false.

$$(I,A)\ |\!-\ \ \textbf{transient}\ \ p\ \ =\ \ \exists\alpha\in A.\ \{\ p\ \}\ \ \alpha\ \ \{\ \neg p\ \}$$

- Now define:

$$\begin{aligned}(I,A)\ |\!-\ \ p\ \ \textbf{ensures}\ \ q\ \ =\ \ & (I,A)\ |\!-\ \ p\ \textbf{unless}\ q \\ & \textit{and}\ (I,A)\ |\!-\ \ \textbf{transient}\ \ p \wedge \neg q\end{aligned}$$

- The *weak fairness* assumption now forces *P* to progress from *p* to *q*. (implying [](p → <>q))

General progress operator

- "ensures" only captures progress driven by a single action. More general progress is expressed by $|\!\rightarrow$ (leads-to).

  It is defined as the *smallest relation* satisfying:

$$\frac{p \textbf{ ensures } q}{p \longmapsto q}$$

(it is a closure of **ensures**)

$$\frac{p \longmapsto q \quad , \quad q \longmapsto r}{p \longmapsto r}$$

(it is transitive)

$$\frac{p_1 \longmapsto q \quad , \quad p_2 \longmapsto q}{p_1 \vee p_2 \longmapsto q}$$

(it is disjunctive on the left argument)

Some other (derived) laws

- $\longmapsto$ itself is relf, trans, and disj.
- Progress – Safety

$$\frac{p \longmapsto q \quad , \quad a \ \textbf{unless}\ b}{p \wedge a \longmapsto (q \wedge a) \vee b}$$

- Bounded progress

$$\frac{p \wedge m{=}C \longmapsto (p \wedge m{<}C) \vee q}{p \longmapsto q}$$

- $0 \leq m$ *holds* innitially
- $0 \leq m$ unless false

Example

- Consider this program, with $x \le 3$ as initial predicate:

$$
\begin{array}{ll}
 & x<3 \;\rightarrow\; x := x+1 \\
[\,] & x<3 \;\rightarrow\; x := x+2 \\
[\,] & x \ge 3 \;\rightarrow\; x := 0
\end{array}
$$

- Notice it has infinite state space.
- Proof of: **true** $\longmapsto$ $x=0$

(1) $4\text{-}x = C$ **ensures** $4\text{-}x<C \lor x=0$

(2) $4\text{-}x = C \longmapsto 4\text{-}x<C \lor x=0$

(3) $4\text{-}x \ge 0$ **unless false**

(4) true $\longmapsto$ $x=0$

Implementing UNITY Logic

- For example, this "proof rule" (actually definition) of **unless**:

$$(\mathit{Init}, A) \;\; |\text{-} \;\; p \;\; \textbf{unless} \;\; q \;\; = \;\; \forall \alpha \in A. \;\; \{\, p \wedge \neg q \,\} \;\; \alpha \;\; \{\, p \vee q \,\}$$

- We can e.g. implement in ML (ala implementation of GCL/wlp), export the resulting verification conditions to HOL for verification.
- We can embed UNITY in HOL itself

Shallow embedding

- Recall this example:

  Define `SEQ $S_1$ $S_2$ *state* = $S_2$ ($S_1$ *state*)` ;

- To cater for non-determinism, represent UNITY actions as relations. So, a thing of type:

  'state $\rightarrow$ 'state $\rightarrow$ bool

- Define `SEQ $S_1$ $S_2$ = ($\lambda s\ u.$ (?$t$. $S_1\ s\ t \ \land\ S_2\ t\ u$))`

Shallow embedding

- How to represent a pre- or post-condition?
- Simply writing e.g. x>0 in HOL is an expression of type bool. How to connect this to the state of an action?
- Represent a state predicate as a function of type ‘state → bool
- For example, is a state is represented by a tuple (x,y), then this is a state predicate: (λ(x,y). x>y)
- And we can define operators like these on state-predicates:

  Define `**AND** p q = (λs. p s /\ q s)`

Shallow embedding

- We define the meaning of Hoare triples in HOL:

  Define `**HOA** $\alpha$ *p q* = (!*s t*. *p s* /\ $\alpha$ *s t* $\Rightarrow$ *q t* )`

- Then we can define the UNITY operator. Below A is an action list, represented as action$\rightarrow$bool.

  Define `**unless** A *p q*
  =
  (!α. MEM α A
  $\Rightarrow$ **HOA** α (p AND NOT q) (p OR q))`

- **transient** and **ensures** can be defined in a similar way.

What do we get?

- Now you can verify UNITY properties in HOL, after properly representing the target program in the corresponding HOL representation.

  Basically, unfolding the definition of UNITY operators reduce a UNITY specification to the underlying predicate logic formulas.

- You can also verify proof rules (of your UNITY logic). For example:

$$\textbf{unless}\ A\ p\ q\ \ /\backslash\ \ \text{valid}\ (q\ \text{IMP}\ r)\ \Rightarrow\ \textbf{unless}\ A\ p\ r$$

Encoding a program

- Let represent a state by a function string → int
- An example of an action:

--\`(λs t. (s x ≥ 3 ⇒ (!var. **if** var=x **then** t var = 0
else t var = s var))
/\
(s x < 3 ⇒ (t = s))) \`--

- You can come up with a DSL to make this less verbose.

Deeper embedding

- For example, by restricting the syntactical structure of actions:

  Hol_datatype `**ACTION** = **Act** of 'pred ⇒ string ⇒ ‘expr`

- sem(**Act** *g x e*)

  $$
  \begin{array}{l}
  = \\
  (\lambda s\ t.\ (g\ s \Rightarrow\ (!var.\ \textbf{if}\ var = x\ \textbf{then}\ t\ var = e\ s \\
  \qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ t\ var = s\ var)) \\
  \qquad \wedge \\
  \qquad (\neg g\ s \Rightarrow (t = s)))
  \end{array}
  $$

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.

λ- calculus

- Grammar:

$$
\begin{aligned}
term \ ::=\ & var \\
\mid\ & const \\
\mid\ & term \quad term && \text{// e.g. } (\lambda x.\ x)\ \ 0 \\
\mid\ & \backslash var.\ \ term && \text{// e.g. } (\lambda x.\ x)
\end{aligned}
$$

- The terms are typed; allowed types:

$$
\begin{aligned}
type \ ::=\ & tyvar && \text{// e.g. } \text{'a} \\
\mid\ & tyconst && \text{// e.g. } bool \\
\mid\ & (type,\ldots,type)\ tyop && \text{// e.g. } bool\ \ list \\
\mid\ & type \rightarrow type
\end{aligned}
$$

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 computation rule

- One single rule called β-reduction

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

- 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:

$$'a \rightarrow\ 'a \rightarrow bool$$

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 “$(\backslash x.\ t)\ u$”

=

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

HOL Primitive logic

$$\text{INST\_TYPE} \quad (\alpha, \tau) \quad (\ |\text{-}\ t \ ) \quad = \quad |\text{-}\ \ t[\tau/\alpha]$$

$$\text{ABS} \ \ (\ |\text{-}\ t=u \ ) \quad = \quad |\text{-} \ \ (\backslash x.\ t) \ = \ (\backslash x.\ u)$$

$$\text{SUBST} \quad (\ |\text{-}\ x=u \ ) \quad t \quad = \quad |\text{-} \ \ t = t[u/x]$$

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 this axiom:

$$\text{ETA\_AX:} \quad \vdash \forall f.\ (\lambda x.\ f\,x) = f$$

HOL Primitive logic

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

$$\text{DISCH} \quad (\ t,\ A\ \mid\!\text{-}\ u\ ) \quad = \quad A \mid\!\text{-}\ t \Rightarrow u$$

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

Examples of building a derived rules

$$\text{UNDISCH} \ (A \ \vdash t \Rightarrow u \ ) \ = \ t, A \ \vdash \ u$$

fun UNDISCH thm_1 = // $A \vdash t \Rightarrow u$

let

t = ... the lhs of the implication in thm_1

thm_2 = ASSUME t // $t \vdash t$

thm_3 = MP thm_1 thm_2 // $t, A \vdash u$

in thm_3 **end**

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

Examples of building a derived rules

$SYM \text{ “}A \ |\!\!- \ t = u\text{”} \quad = \quad A \ |\!\!- \ u = t$

```
fun  SYM  thm1  =                                   //      A  |-  t = u
  let
  t  =  ....  the lhs of the equality in thm1
  thm2  =  REFL t                                   //         |-  t = t
  thm3  =  SUBST { “x” → thm1 } “x=c”  thm2         //      A  |-  u=t

  in  thm3  end
```

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 → captured by this definition:

$$\text{T\_DEF:} \quad |\!- \quad T \;=\; ((\lambda x{:}bool.\ x) \;=\; (\lambda x.\ x))$$

Encoding Predicate Logic
(Desc 3.2)

- Introduce constant "∀ "of type ('a→bool)→bool, defined as follows:

$$\text{FORALL\_DEF}: \quad \vdash \quad \forall P \quad = \quad (P = (\lambda x.\ T))$$

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

- Now we define "F" as follows:

$$\text{F\_DEF}: \quad \vdash \quad F \;=\; \forall t{:}bool.\ t$$

- Puzzle for you: prove just using HOL primitive rules (more later) that ¬(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 : $\vdash \exists P \ = \ P\, @P$

And some axioms ...

- BOOL_CASES_AX: $\quad \vdash \forall b.\ (b=T) \lor (b=F)$

- IMP_ANTISYM:

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

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 // $\vdash \sim(T{=}F)$

And this infinity axiom...

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

INFINITY_AX :

$$|\text{-}\ \exists f{:}ind \rightarrow ind.\ \ One\_One\, f\ \ \wedge\ \ \sim Onto\, f$$

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

One One f = ∀x y. (f x = f y) ⇒ (x = y) // every point in rng f has at most 1 source
Onto f = ∀y. ∃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

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 $`RGB = RED \mid GREEN \mid BLUE`$

Hol_datatype $`MyBinTree = Leaf\ \ int \mid Node\ \ MyBinTree\ \ MyBinTree$

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 a new type by postulating it.

- 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)   …
`--) ;

Similarly:

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

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

- We declare a new type "num", and declare its constructors:
  - $0 : num$
  - $SUC : num \rightarrow num$

- We also need some axioms, e.g. Peano's :

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

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

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

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.

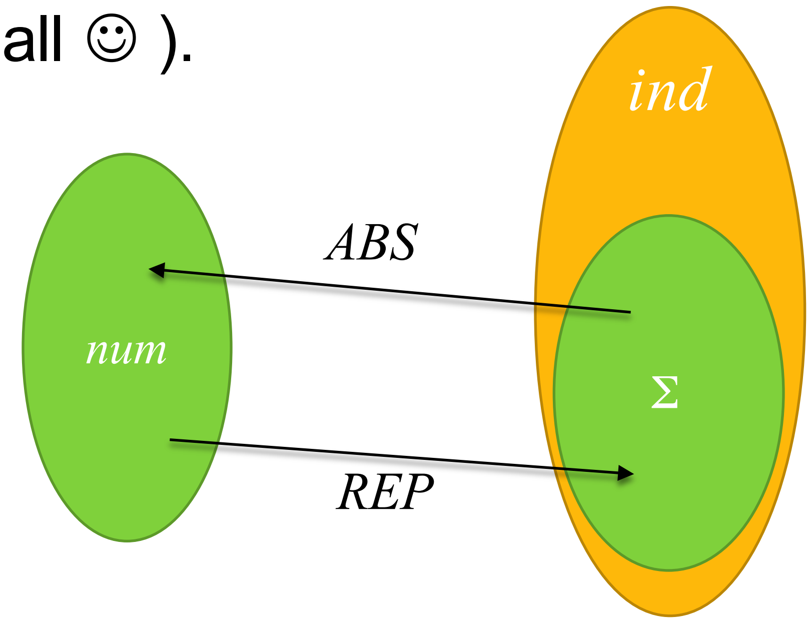
- Contradicting type axioms imply that your type τ is actually empty. So, e.g. β-reduction should <u>not</u> be possible:

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

  However HOL assumes 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 σ as the function f:ind→ind that INFINITY_AX says to exist. So, f satisfies:

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

- Take σ is "the model" of SUC at the ind-side.
- Similarly, model 0 by Z, defined by:

$$\boxed{Z = @(\lambda z{:}ind.\ {\sim}(\exists x.\ z = \sigma\, x))}$$

So, Z is some member of "ind" who has no f-source (or σ source).

The Σ part

- Define Σ as a subset of ind that admits num-induction. Prove it is not empty.
  We'll encode Σ as a predicate ind→bool:

$$\Sigma\, x \;=\; (\forall P.\; \underbrace{P\, Z \;\wedge\; (\forall y.\, P\, y \Rightarrow P\,(\sigma\, y))}_{\text{Let's call such a "set" } P \text{ as a num-inductive set.}} \;\Rightarrow\; P\, x)$$

- So, Σ is the smallest num-inductive set.
- You get the num-induction principle on Σ.

Defining "num"

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

$$rep : num \to ind$$
$$abs : ind \to num$$

- Then add these axioms:

$$
\begin{array}{lcl}
rep\ 0 & = & Z \\
rep\ (SUC\ n) & = & \sigma\ (rep\ n)
\end{array}
$$

$$(\forall n{:}num.\ \ \Sigma\,(rep\ n)\ )$$

$$(\forall n{:}num.\ \ abs(rep\ n) \ = n\ )$$

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

Now you can actually prove the orginінal axioms of num

- E.g. to prove $0 \neq$ SUC n; we prove this with contradiction:

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

Automated

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

$$Hol\_datatype\ `NaturalNumber = ZERO \mid NEXT\ of\ NaturalNumber$$

will generate e.g :

$$NaturalNumber\_distinct: \quad \vdash\ \forall n.\ \sim(ZERO = NEXT\ n)$$

$$NaturalNumber\_induction:$$

$$\vdash\ \forall P.\quad P\ ZERO\ /\backslash\ (\forall n.\ P\ n \Rightarrow P\ (NEXT\ n)) \Rightarrow (\forall \mathrm{n}.\ P\ n)$$

Manipulating Terms

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: $assumptions \;\; ?\text{-} \;\; ok\ 10$

Contain $“(\forall n.\ \ P\ n \Rightarrow ok\ n)”$

$assumptions \;\; ?\text{-} \;\; P\ 10$

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

But how?? With the tactic below:

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

$FIRST\_ASSUM \quad MATCH\_MP\_TAC$

$“\ assumptions \;\; ?\text{-} \;\; ok\ 10”$

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

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:

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

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 → $\vdash\ t = u$
- Type: $conv\ =\ term \rightarrow thm$ such that if *c:conv*

  then *c t* can produce $\vdash\ t = something$

- 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” →

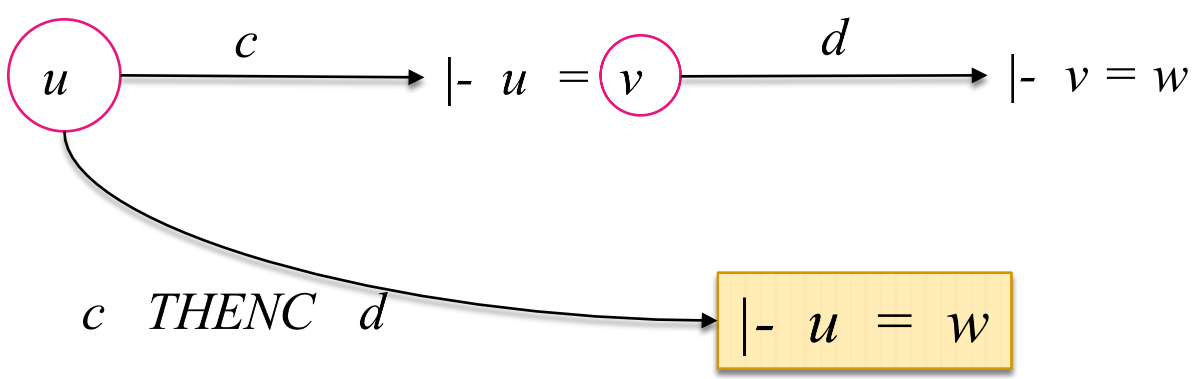
|- (f=g) = (!x. f x = g x)

Composing conversions

- The unit and zero: ALL_CONV, NO_CONV

- Sequencing: $c \quad THENC \quad d$

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 ...

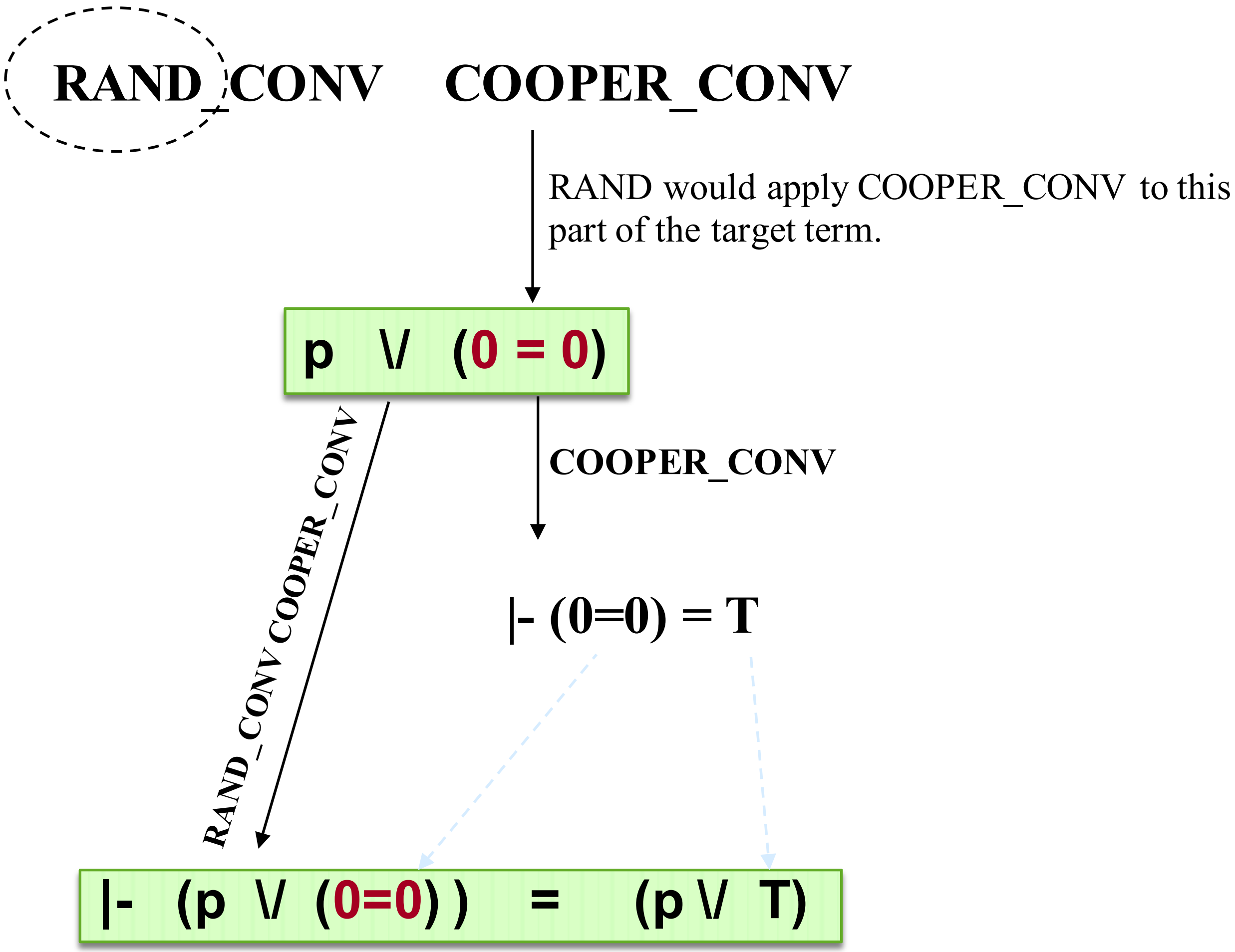
- Allows conversion to be applied to specific subtrees instead of the whole tree:

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

$RAND\_CONV\ c\ t$ 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: $?\text{-} \quad \sim (f = g)$

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

CONV_TAC (RAND_CONV FUN_EQ_CONV)