Preuves Interactives et Applications

Christine Paulin & Burkhart Wolff

http://www.lri.fr/ ~paulin/PreuvesInteractives

Université Paris-Saclay

HOL and its Specification Constructs

Revision: Documents and Commands

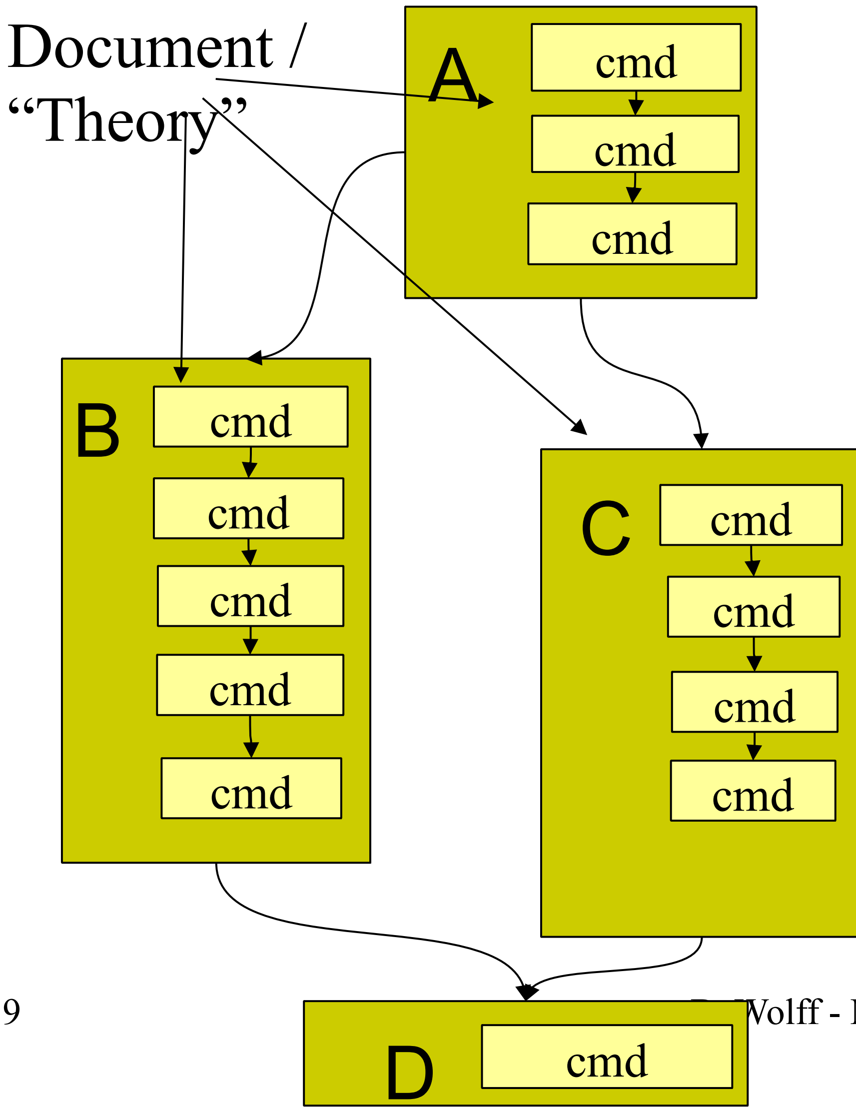
- Isabelle has (similar to Eclipse) a „document-centric" view of development:

  there is a notion on an entire "project" which is processed globally.

- Documents (~ projects in Eclipse) consists of files (with potentially different file-type); .thy files consists of headers commands.

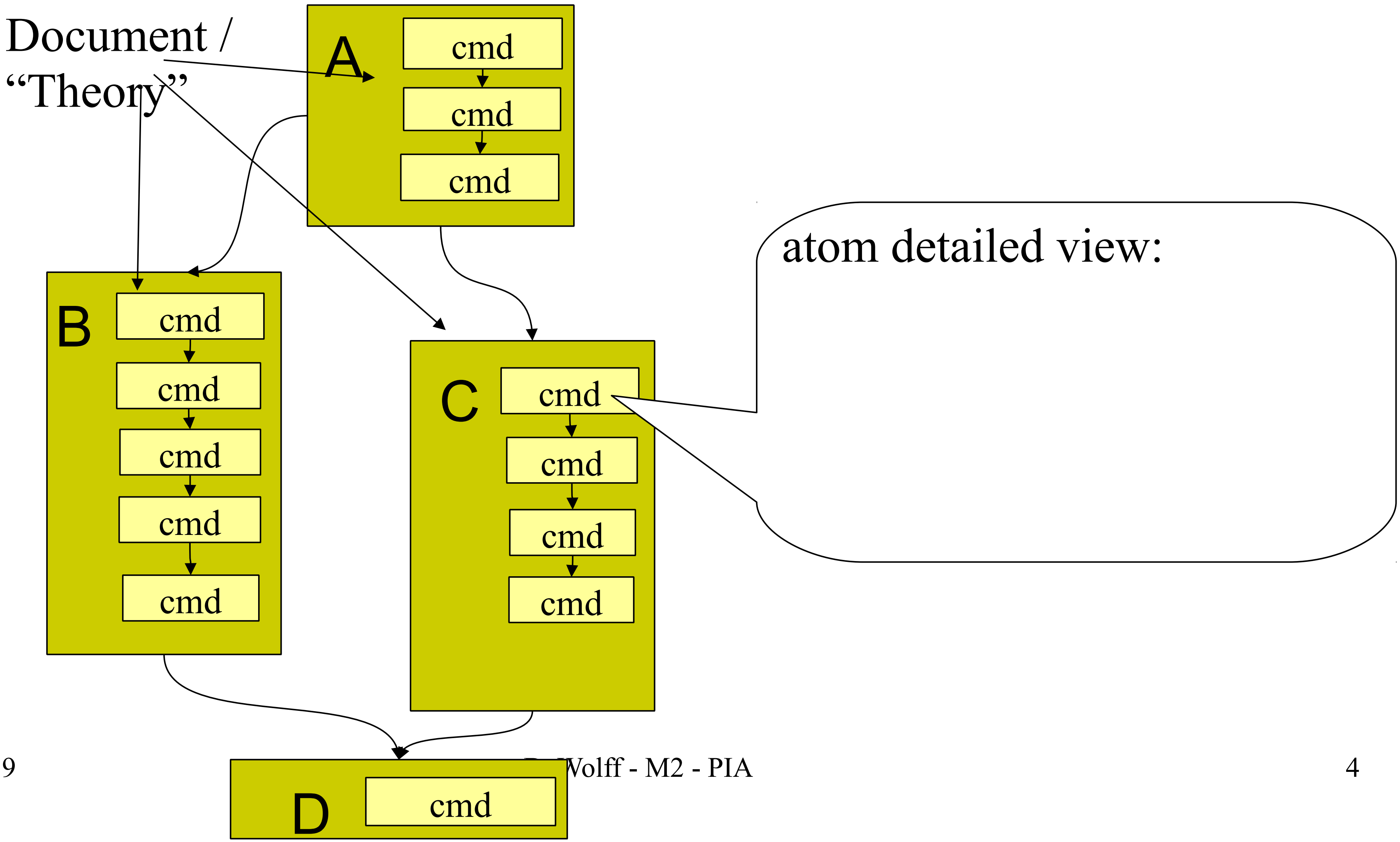
What is Isabelle as a System ?

- Global View of a “session“



What is Isabelle as a System ?

- Global View



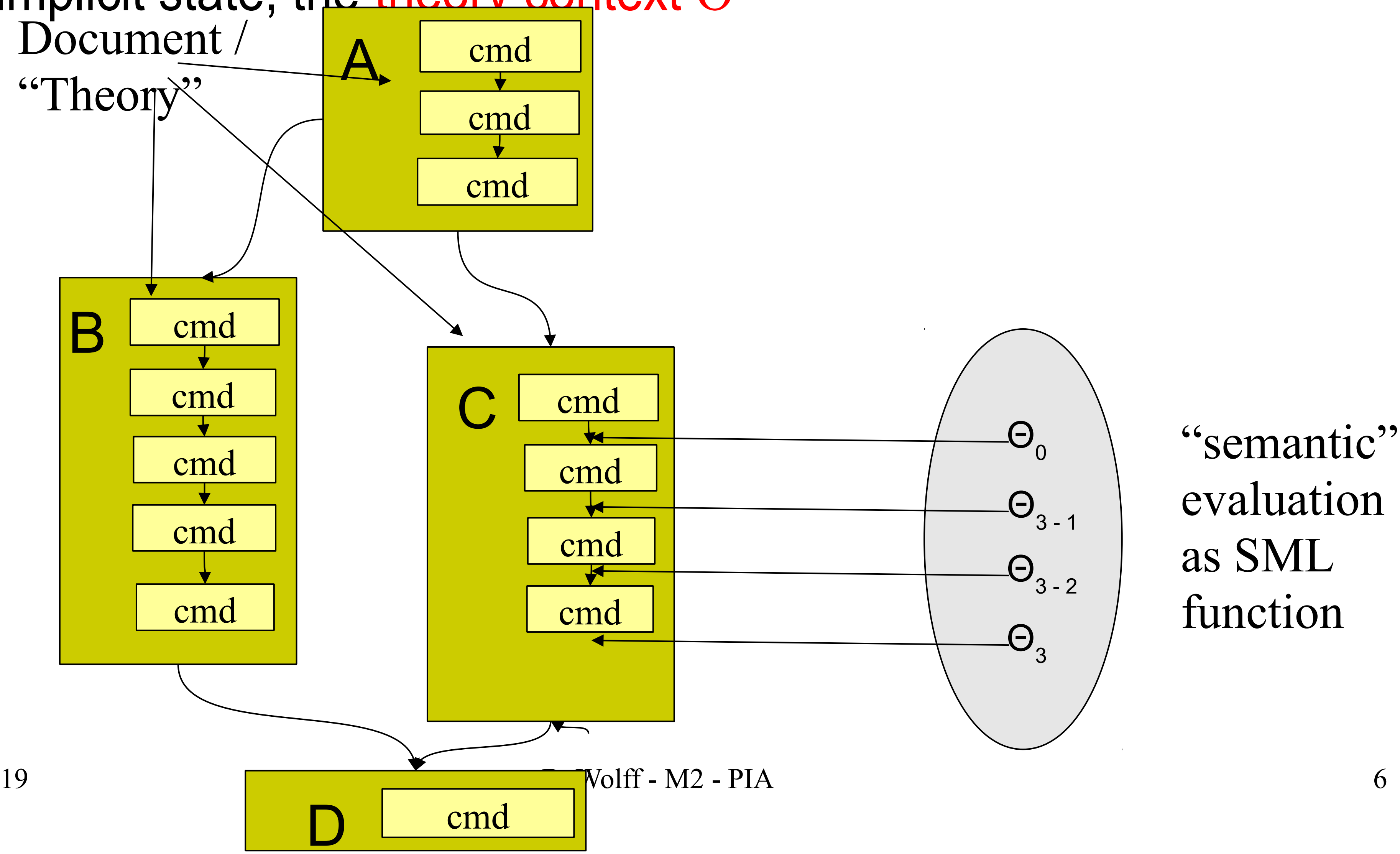
Revision: Documents and Commands

- Each position in document corresponds
  - to a "global context" Θ
  - to a "local context" Θ, Γ

- There are specific „Inspection Commands“ that give access to information in the contexts
  - thm, term, typ, value, prop : global context
  - print_cases, facts, ... , thm : local context

What is Isabelle as a System ?

- Document “positions” were evaluated to an implicit state, the theory context Θ



Inspection Commands

- Validating a type-expression:

typ “<hol-typ>”

example: typ "('a × 'β ⇒ bool)set"

- Type-checking terms:

term “<hol-term>”

example: term “(a::nat) + b = b + a"

Inspection Commands

- Type-checking propositions:

prop “<boolean-term>”

example: prop “∃t. ∀u. H t u ⟶ ¬ Q u"

Note: Propositions may contain free variables, which are implicitly universally quantified!

- Checking Theorem Names, Printing Theorems:

thm “<theorem-id>”

example: thm refl sym subst

Search Commands

- Searching theorem id's in the global context:

find_theorems "<pattern-list>"

example: find_theorems name:"HOL" "_ = _"

Text Commands

- Text-Commands:

chapter ‹ text ›

section ‹ text ›

subsection ‹ text ›

text ‹ text ›

- Text: spell-checked
- may contain „antiquotations“ for types, terms, thms, code, ...

Code-execution Commands

- Evaluating terms:

```
value “<hol-term>”
```

example: value “(3::nat) + 4 = 7"

- Code-Generation:

```
export_code “<hol-id>” in “<lang>”
 module_name “<sml-id>” file “<path>”
```

example: export_code odometer_function_step in SML
module_name Odo_Function file "code/sml/odo.sml"

Basic Declaration Commands

- Type Declaration

typedecl “(α_1,...,α_n) <typconstructor-id>”

example: typedecl “L"

- (Unspecified) Constant Declaration:

consts c :: „τ“

example: consts True :: “bool”

Simple Proof Commands

- Simple (Backward) Proofs:

```
lemma  <thmname> :
  [<contextelem>+ shows] "<phi>"
   <proof>
```

There are different formats of proofs, we concentrate on the simplest one:

apply(<method$_1$>) ... apply(<method$_n$>) done

Exercise demo3.thy

- Examples

lemma X1 : “A ⟹ B ⟹ C ⟹ (A ∧ B) ∧ C”

(* output: ⟦A; B; C⟧ ⟹ (A∧B)∧C) *)

lemma X2 : assume “A” and “B” and “C”

shows “(A ∧ B) ∧ C”

lemma X2 : assume h1: “A” and h2: “B” and h3: “C”

shows “(A ∧ B) ∧ C”

Basic Backward Proof Methods

- The most elementary proof method is the rule <thmname> method. It is used for introduction rules.

```
apply(rule <thm>)
```

It basically proceeds in two phases:

- it searches the <thm> and replaces the free variables by fresh variables of the form ?X,?Y,?Z (schematic variables)
- it constructs an instance of <thmname> by unification; this means that the conclusion of <thmname> must finally match (modulo β and α red.) against the conclusion of the current (first) goal.

examples: apply(rule impI) apply(rule iffI)
apply(rule deMorgan[symmetric])

Basic Backward Proof Methods

- The user can help the unification process by giving a (partial) substiutution:

apply(rule_tac <*subst*> in <thm>)

… where <*subst*> is of the form: x_1="ϕ_1" and x_n="ϕ_n"
and the x_i are some variables of <thm>

example: apply(rule_tac P="λX. H(c + b) (X)" in subst, rule add.commute)

Converts goal "H(c + b) (3 + Suc a)" into
goal „H (c + b) (Suc a + 3)"

Basic Backward Proof Methods

- The most elementary proof method is the rule <thmname> method. It is used for elimination rules.

apply(erule <thm>)

It basically proceeds in two phases:

- it searches the <thm> and replaces the free variables by fresh variables of the form ?X,?Y,?Z (schematic variables)
- it constructs an instance of <thmname> by unification; this means that the conclusion of <thmname> must finally match (modulo β and α red.) against a premise of the current (first) goal.

examples: apply(erule impE) apply(erule allE)

Basic Backward Proof Methods

- The user can help the unification process by giving a (partial) substitution:

```
apply(erule_tac <subst> in <thm>)
```

… where <*subst*> is of the form: x_1="ϕ_1" and x_n="ϕ_n"
and the x_i are some variables of <thm>

example: apply(rule_tac P="λX. H(c + b) (X)" in subst, rule add.commute)

Converts goal "H(c + b) (3 + Suc a)" into
goal „H (c + b) (Suc a + 3)"

Basic Backward Proof Methods

- Closing a goal

apply(assumption)

unifies a premisse against a conclusion.

example: apply(assumption)

Closes goal "$H(c + 3) \Longrightarrow H(c + ?b)$"
and propagates the substitution $?b \mapsto 3$
throughout the proof state.

Basic Forward Proof Methods

- chaining by modus-ponens:
  - <thm>[THEN <thm>]
    example: add.commute[THEN subst]
    produces ?P (?a1 + ?b1) ⟹ ?P (?b1 + ?a1)
  - <thm>[OF <thm>]
    example: add.commute[THEN subst]
    produces the same
  - <thm>[symmetric]
    example: de_Morgan_conj[symmetric]
    produces (¬ ?P ∨ ¬ ?Q) = (¬ (?P ∧ ?Q))
- instantiation with consts or free variables:
  - <thm>[of <term> <term> <term>]
    example: add.commute[of _ "3"] produces ?a + 3 = 3 + ?a

At a Glance

- low-level methods (without substitution)
  - assumption (unifies conclusion vs. a premise)
  - rule[_tac <*subst*> in] <thmname>
    PROLOG - like resolution step using HO-Unification
  - erule[_tac <*subst*> in] <thmname>
    elimination resolution (for ND elimination rules)
  - subst <thmname>
    does one rewrite-step
    (by instantiating the HOL subst-rule)
  - drule[_tac <*subst*> in] <thmname>
    destruction resolution (for ND destruction rules)

Specification Commands

- Simple Definitions (Non-Rec. core variant):

definition f::"<τ>" where "f x_1 … x_n = <t>"

example: definition C::"bool ⇒ bool"

where "C x = x"

- Type Definitions:

typedef ('α_1..'α_n) κ = "<set-expr>" <proof>

example: typedef even = "{x::int. x mod 2 = 0}"

Isabelle Specification Constructs

- Major example:

  The introduction of the cartesian product:

```
subsubsection {* Type definition *}

definition Pair_Rep :: "'a ⇒ 'b ⇒ 'a ⇒ 'b ⇒ bool"
where    "Pair_Rep a b = (λx y. x = a ∧ y = b)"


definition "prod = {f. ∃ a b. f = Pair_Rep (a :: 'a) (b :: 'b)}"

typedef ('a, 'b) prod (infixr "*" 20) = "prod :: ('a ⇒ 'b ⇒ bool) set"
                                     unfolding prod_def by auto

type_notation (xsymbols) "prod" ("(_ ×/ _)" [21, 20] 20)
```

Specification Mechanism Commands

- Datatype Definitions (similar SML):
  (Machinery behind : complex series of const and typedefs !)

```
datatype ('a1..'an) Θ =
 <c> :: "<τ>" | ... | <c> :: "<τ>"
```

- Recursive Function Definitions:
  (Machinery behind: Veeery complex series of const and typedefs and automated proofs!)

```
fun <c> ::"<τ>" where
    "<c> <pattern> = <t>"
  | ...
  |  "<c> <pattern> = <t>"
```

Specification Mechanism Commands

- Datatype Definitions (similar SML):
  (Machinery behind : complex !)

```
datatype ('a1..'an) Θ =
 <c> :: "<τ>"  | ... | <c> :: "<τ>"
```

- Recursive Function Definitions:
  (Machinery behind: Veeery complex!)

```
fun <c> ::"<τ>" where
    "<c> <pattern> = <t>"
 | ...
 |  "<c> <pattern> = <t>"
```

NOTE: Isabelle HOL compiles this internally to axiomatic definitions, i.e. a "model" in HOL!!!

Specification Mechanism Commands

- Datatype Definitions (similar SML):
  Examples:

```
datatype mynat = ZERO | SUC mynat

datatype 'a list = MT | CONS "'a" "'a list"
```

Specification Mechanism Commands

- Inductively Defined Sets:

```
inductive     <c> [ for <v>:: "<τ>" ]
  where  <thmname> : "<ϕ>"
              | ...
              | <thmname> = <ϕ>
```

```
example: inductive path for rel ::"'a ⇒ 'a ⇒ bool"
         where  base : "path rel x x"
           |    step : "rel x y ⟹ path rel y z ⟹ path rel x z"
```

Specification Mechanism Commands

- Extended Notation for Cartesian Products: records (as in SML or OCaml; gives a slightly OO-flavor)

```
record    <c> = [<record> + ]
  tag₁ :: "<τ₁>"
  ...
  tagₙ :: "<τₙ>"
```

- ... introduces also semantics and syntax for
  - selectors : $tag_1\ x$
  - constructors : $⦇ tag_1 = x_1, \ldots, tag_n = x_n ⦈$
  - update-functions : $x ⦇ tag_1 := x_n ⦈$

More on Proof-Methods

- Some composed methods (internally based on assumption, erule_tac and rule_tac + tactic code that constructs the substitutions)
  - subst <equation>

    (one step left-to-right rewrite, choose any redex)
  - subst <equation>[symmetric]

    (one step right-to-left rewrite, choose any redex)
  - subst (<n>) <equation>
    (one step left-to-right rewrite, choose n-th redex)

More on Proof-Methods

- Some composed methods (internally based on assumption, erule_tac and rule_tac + tactic code that constructs the substitutions)
  - simp

    (arbitrary number of left-to-right rewrites, assumption or rule refl attepted at the end; a global simpset in the background is used.)
  - simp add: <equation> ... <equation>

More on Proof-Methods

- Some composed methods (internally based on assumption, erule_tac and rule_tac + tactic code that constructs the substitutions)
  - auto
    (apply in exaustive, non-deterministic manner:
    all introduction rules, elimination rules and
  - auto intro: <rule> ... <rule>
    elim: <erule> ... <erule>
    simp: <equation> ... <equation>

More on Proof-Methods

- Some composed methods (internally based on assumption, erule_tac and rule_tac + tactic code that constructs the substitutions)
  - cases „<formula>“
    (split top goal into 2 cases:
    <formula> is true or <formula> is false)
  - cases „<variable>“
    (- precondition : <variable> has type t which is a data-type)
    search for splitting rule and do case-split over this variable.
  - induct_tac „<variable>“
    (- precondition : <variable> has type t which is a data-type)
    search for induction rule and do induction over this variable.

Screenshot with Examples