Preuves Interactives et Applications

Christine Paulin & Burkhart Wolff

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

Université Paris-Saclay

HOL and its Specification Constructs

Revisions

Revision: Documents and Commands

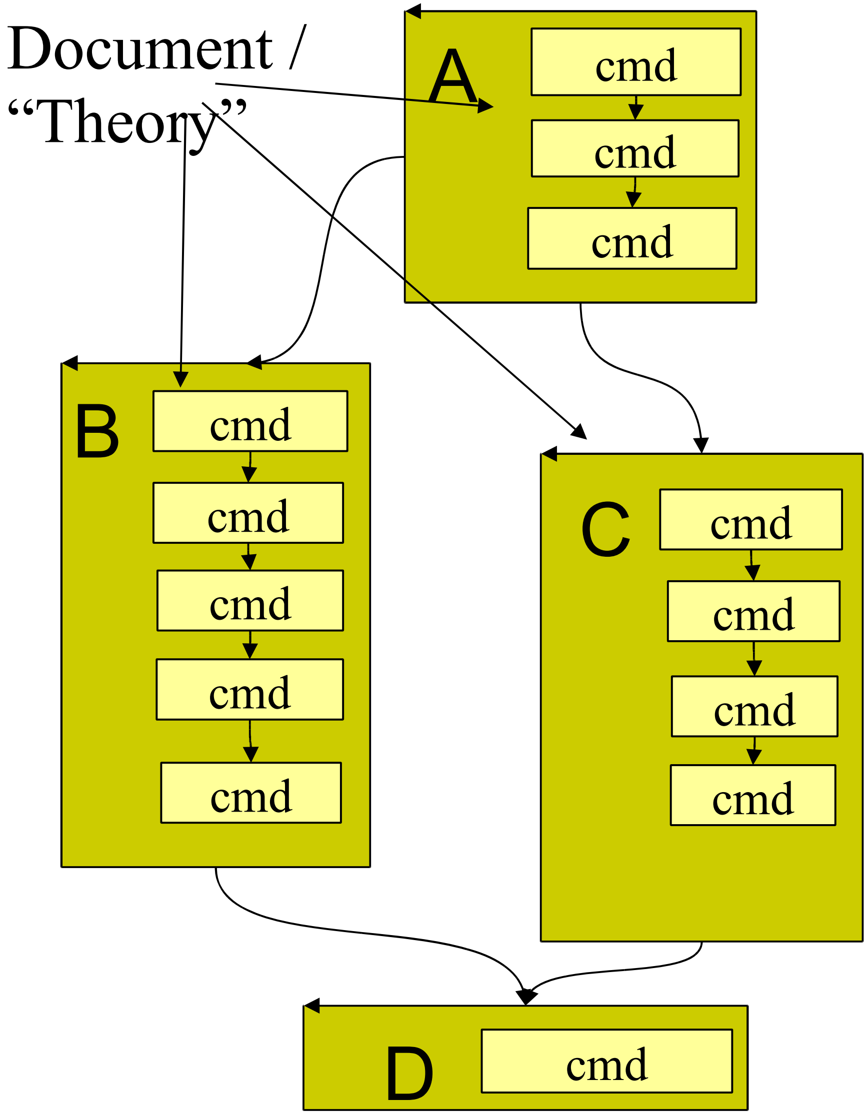
- Isabelle has (similar to Eclipse) a „document-centric" view of development:

  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.

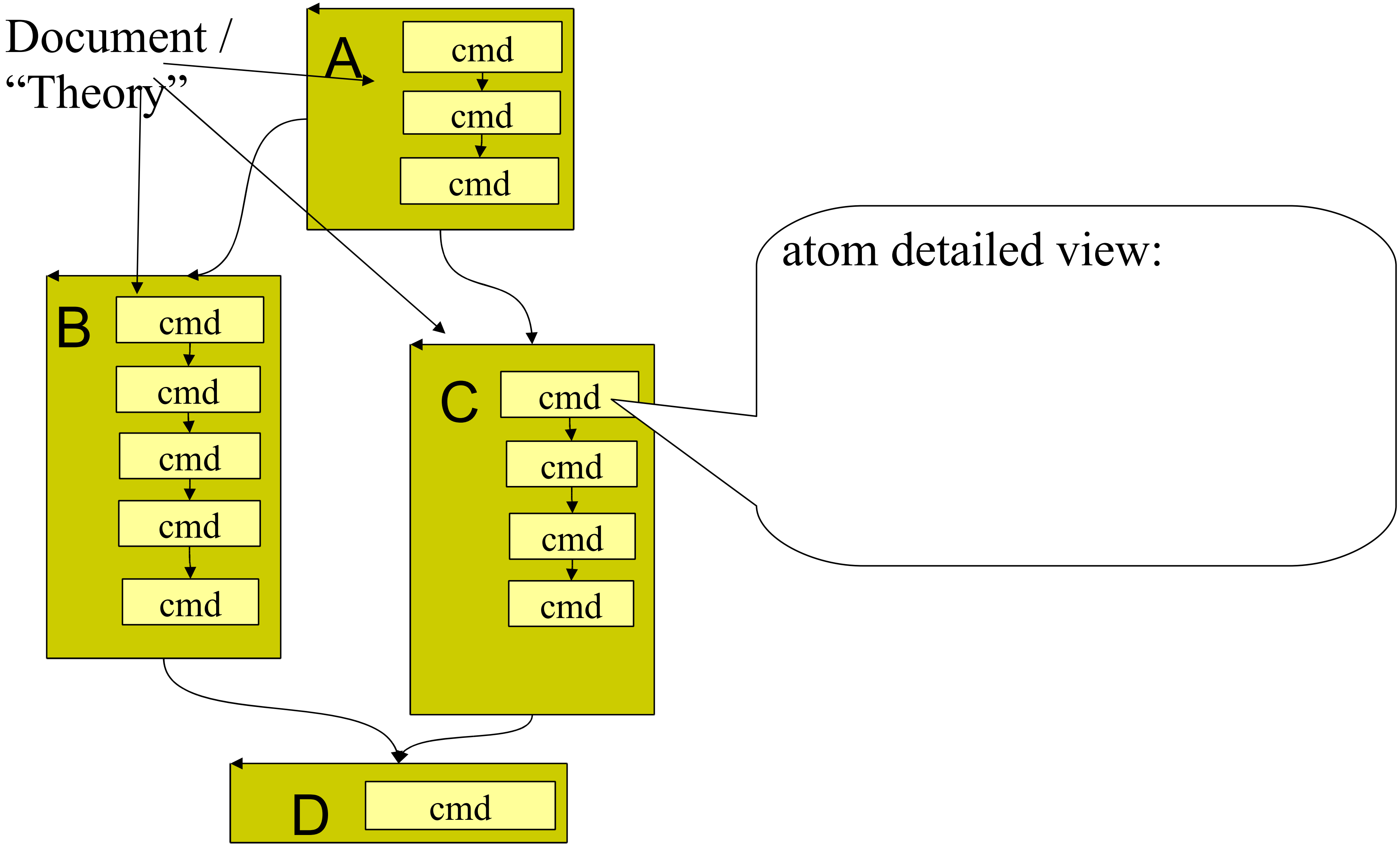
What is Isabelle as a System ?

- Global View of a “session“



What is Isabelle as a System ?

- Global View



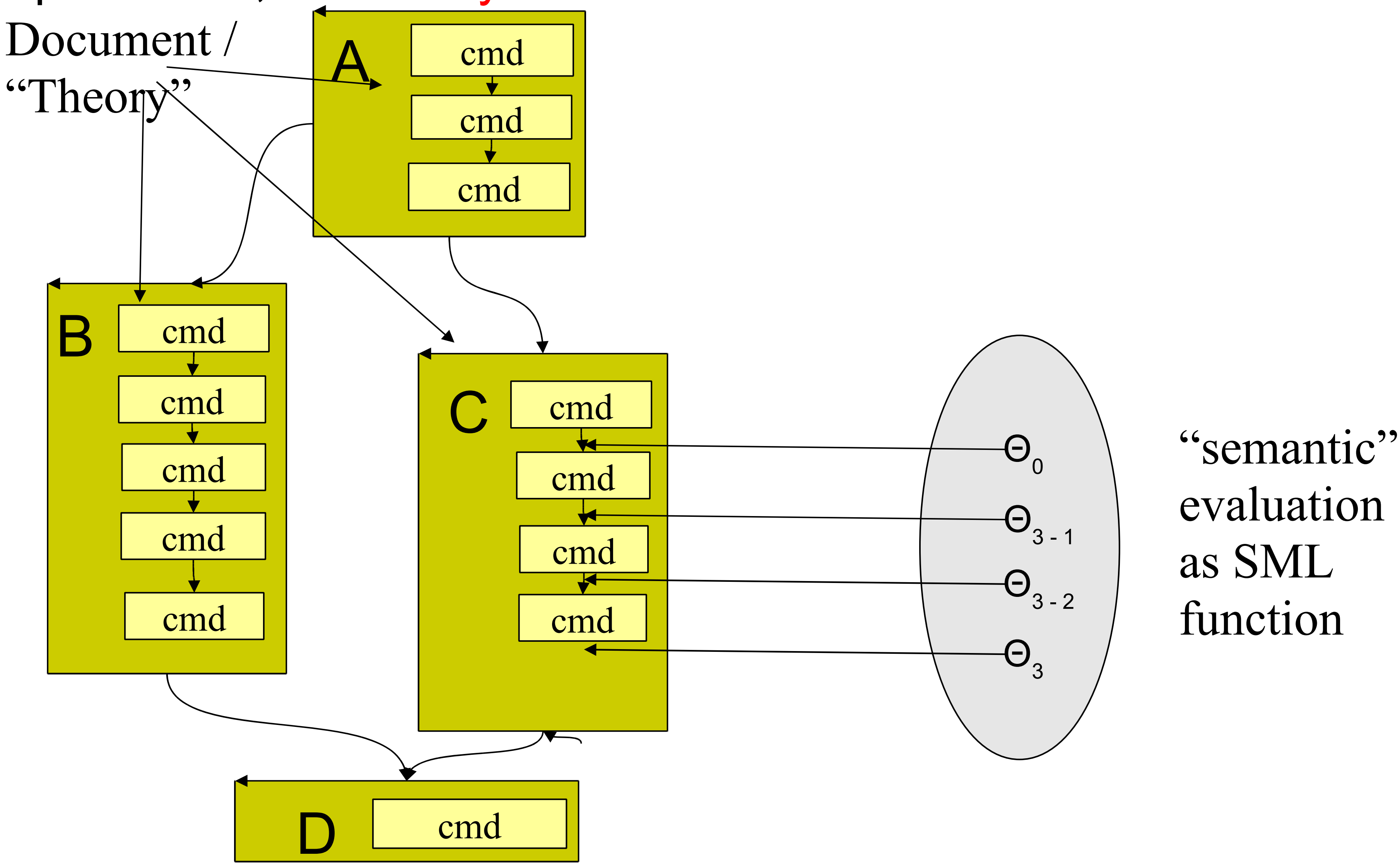
Revision: Documents and Commands

- 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

- Type-checking terms:

term “<hol-term>”

example: term “(a::nat) + b = b + a"

- Evaluating terms:

value “<hol-term>”

example: term “(3::nat) + 4 = 7"

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 : “$A \Longrightarrow B \Longrightarrow C \Longrightarrow (A \land B) \land C$”

(* output: $[\![A; B; C]\!] \Longrightarrow (A \land B) \land C$) *)

lemma X2 : assume “A” and “B” and “C”

shows “$(A \land B) \land C$”

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

shows “$(A \land B) \land C$”

How to built theories in a logically safe manner ?

- Beyond the question:

  Is the Kernel of Isabelle correct ?

  there is the question:

  Is the HOL Library consistent ?

  What guarantees can we have for systems with 15000 rules ???

Isabelle Specification Constructs

- Constant Definitions:

```
definition f::"<τ>"
  where <name> : "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

and

Theory Construction by Conservative Extension

Semantics of „Constant Definition"

- Constant definition:

  (Σ, A) "∈" Θ

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

  $(\Sigma \oplus f::\tau\,,\ A \oplus \{f\_def \mapsto \text{"}f\ x_1 \ldots x_n = \text{<expr>"}\})$ "∈" Θ'

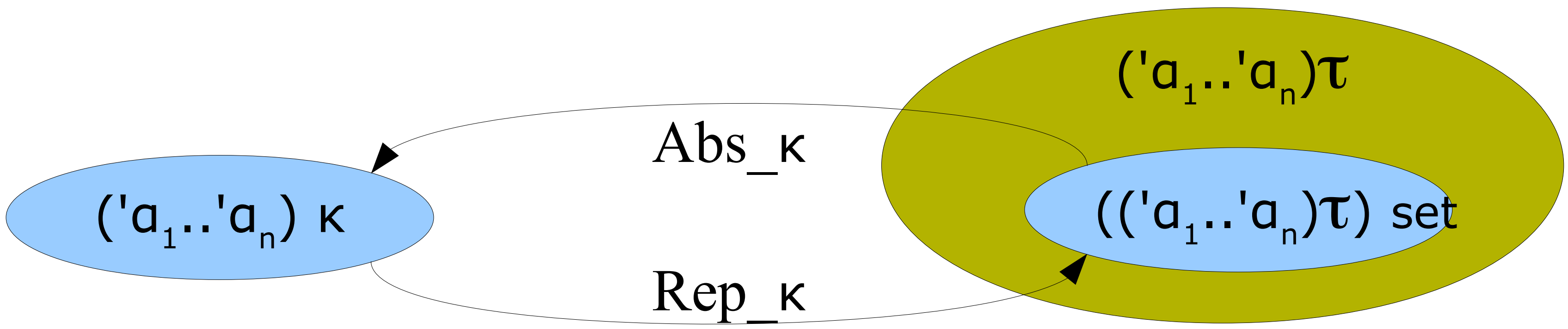
  - where f is "fresh" in Θ
  - $\lambda\ x_1 \ldots x_n$ = <expr> is closed [and type-closed]
  - f does not occur in <expr>

Semantics of a „Type Definition“

- Idea: Similar to constant definitions; we define the new entity (“a type”) by an old one.
- For Type Definitions, we define the new type to be isomorphic to a (non-empty) subset of an old one.
- The Isomorphism is stated by three (conservative) axioms.

Semantics of a „Type Definition"

- Idea: Similar to constant definitions; we define the new entity ("a type") by an old one.



Isabelle Specification Constructs

- Type definition:

(Σ, A) "$\in$" Θ

typedef $('\alpha_1..'\alpha_n)\ \kappa$ =
"<expr:: $(('\alpha_1..'\alpha_n)\tau)$ set>" <proof>

$$(\Sigma \oplus ('\alpha_1..'\alpha_n)\ \kappa \oplus Abs\_\kappa::('\alpha_1..'\alpha_n)\tau \Rightarrow ('\alpha_1..'\alpha_n)\kappa$$
$$\oplus\ Rep\_\kappa::('\alpha_1..'\alpha_n)\kappa \Rightarrow ('\alpha_1..'\alpha_n)\tau$$
$$A \oplus \{Rep\_\kappa\_inverse \mapsto Abs\_\kappa\ (Rep\_\kappa\ x) = x\}$$
$$\oplus \{Rep\_\kappa\_inject \mapsto (Rep\_\kappa\ x = Rep\_\kappa\ y) = (x = y)\}$$
$$\oplus \{Rep\_\kappa \mapsto Rep\_\kappa\ x \in \{x.\ expr\ x\})\ \text{"}\in\text{"}\ \Theta'$$

- where the type-constructor κ is "fresh" in Θ
- expr is closed
- <expr:: $('\alpha_1..'\alpha_n)\tau$ set> is non-empty (to be proven by a witness)

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 ('a_1..'a_n) 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

- Inductively Defined Sets:

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

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

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

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