Preuves Interactives et Applications

Burkhart Wolff

http://www.lri.fr/~wolff/teach-material/2017-18/M2-CSMR/U3-Theory-Extensions.pdf

Université Paris-Saclay

HOL Foundations and Specification Constructs

Revision: Documents and Commands

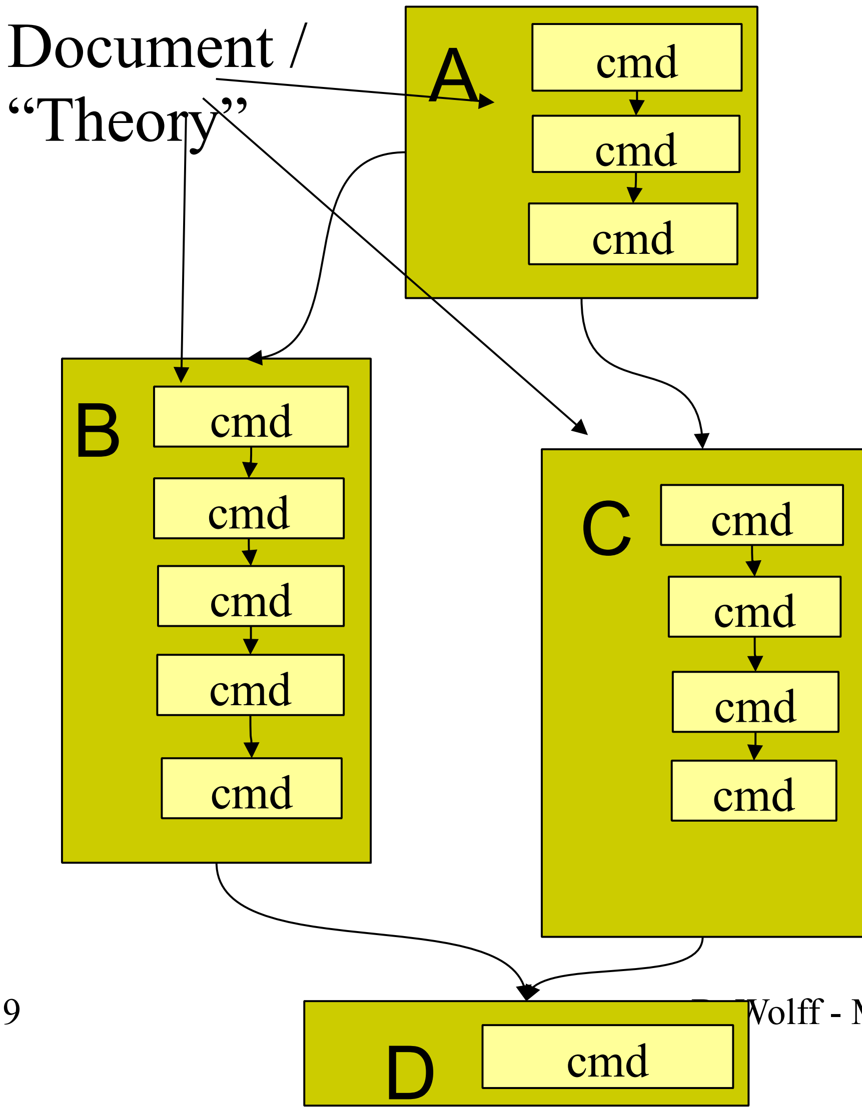
- Isabelle has (similar to Eclipse) a „document-centric" view of development:

  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“



Revision: Documents and Commands

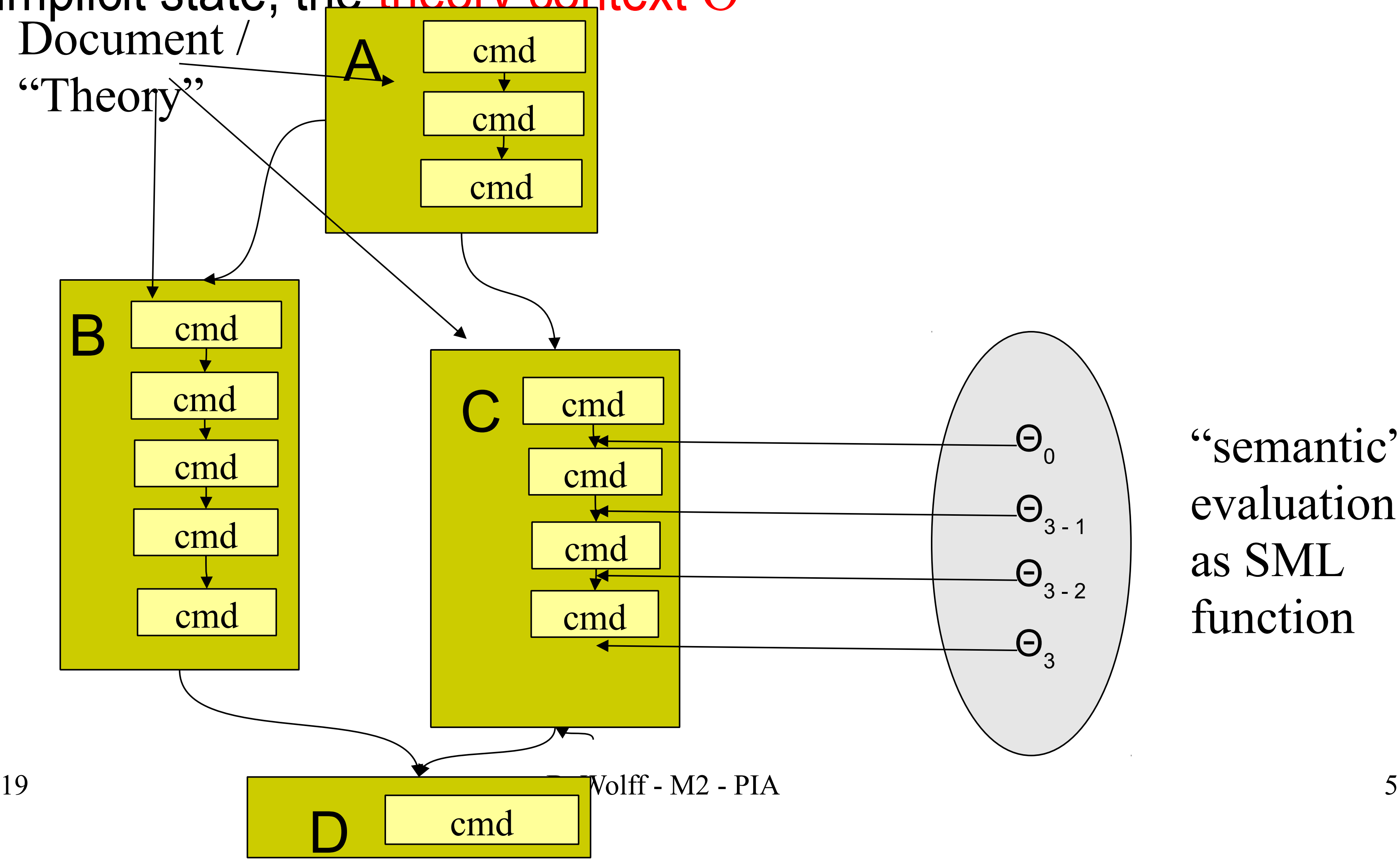
- Each position in document corresponds
  - to a “global context” Θ
    (containing a signature Σ and a set of axioms A)
  - to a “local context” Θ, Γ

- 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



Recall: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" (NOT Isabelle/HOL)

Recall:Basic Declaration Commands

- Constant Declaration “Semantics”:

$$(\Sigma, A) \text{"}\in\text{"}\ \Theta$$

consts c :: „τ“

$$(\Sigma \oplus (C \mapsto \tau)\ , A) \text{"}\in\text{"}\ \Theta'$$

- where the constant c is “fresh” in Σ

How to built theories in a logically safe manner ?

- Beyond, there are a number of questions:
  - Is the logic HOL consistent ?
  - Is HOL correctly implemented in Isabelle ?
  - How to extend HOL in a logically safe way ?

To the HOL library „Main“, for example ?

How to built theories in a logically safe manner ?

- Beyond, there are a number of questions:
  - Is the logic HOL consistent ?
  - Is HOL correctly implemented in Isabelle ?
  - How to extend HOL in a logically safe way ?

To the HOL library „Main“, for example ?

We will address these questions one by one ...

How to built theories in a logically safe manner ?

- HOL consistency
  - ... can only be answered relatively,
    i.e. relative to a logical system which gives
    a formal „interpretation" of HOL terms.
  - the gold-standard for mathematicians and
    logicians is „Zermelo-Fraenkel Set Theory"
    plus „axiom of choice", called ZFC.
  - it is possible to give several interpretations of HOL
    in ZFC and prove the validity of HOL-s core
    axioms relative to these interpretations.

How to built theories in a logically safe manner ?

- HOL consistency
  - ZFC gives a kind of „universe of sets" V with the properties:
    - an infinite set I is part of V
    - any product V'× V" is part of V, if V' and V" are
    - any potence set $\mathcal{P}$(V') is part of V provided that V' is. (this is not possible in a typed set-theory)
  - Since relations $\mathcal{P}$(V'× V") are part of V, it is possible to express in V function spaces.

How to built theories in a logically safe manner ?

- HOL consistency
  - ZFC gives a kind of „universe of sets“ V with the properties:
    - an infinite set I is part of V
    - any product V‘× V“ is part of V, if V‘ and V“ are
    - any potence set $\mathcal{P}$(V‘) is part of V provided that V‘ is. (this is not possible in a typed set-theory)
  - Since relations $\mathcal{P}$(V‘× V“) are part of V, it is possible to express in V function spaces.

How to built theories in a logically safe manner ?

- HOL consistency
  - Since relations $\mathcal{P}(V' \times V'')$ are part of V, it is possible to express in V function spaces:
    - $A \Rightarrow_{standard} B = \{f: \mathcal{P}(V' \times V'') \mid f \neq \emptyset \text{ and } f \text{ is function}\}$
    - $\emptyset \neq (A \Rightarrow_{henkin} B) \subseteq \{f: \mathcal{P}(V' \times V'') \mid f \neq \emptyset \text{ and } f \text{ is function}\}$
    - $A \Rightarrow_{construct} B = \{f: \mathcal{P}(V' \times V'') \mid f \neq \emptyset \text{ and }$
      $f \text{ is a computable function}\}$
  - On this basis, we can give a standard (Henkin-style, constructivist) interpretation of HOL types into V

$$I_{standard} : \tau \Rightarrow V,\ I_{henkin} : \tau \Rightarrow V,\ I_{construct} : \tau \Rightarrow V$$

How to built theories in a logically safe manner ?

- HOL consistency
  - On this basis, we can give a standard interpretation of HOL core types into V
    - $I_{standard}$ $[\![bool]\!] = \{a,b\}$ (where a,b are some distinct elements from the infinite set I)
    - $I_{standard}$ $[\![ind]\!] = I'$
    - $I_{standard}$ $[\![\tau \Rightarrow \tau']\!] =$ $I_{standard}$ $[\![\tau]\!] \Rightarrow_{standard}$ $I_{standard}$ $[\![\tau']\!]$
  - It is easy to show that our typing rules are consistent with $I_{standard}$, I_{henkin}, $I_{construct}$.

How to built theories in a logically safe manner ?

- HOL consistency
  - Core HOL needs a small number of axioms.
  - Traditional papers [Andrews86] reduce it to 6 axioms plus the axiom of infinity:

$$\exists\ f{::}ind \Rightarrow ind.\ injective\ f \wedge \neg surjective\ f$$

  - The presentation in Isabelle/HOL looks as follows:

How to built theories in a logically safe manner ?

- The presentation in Isabelle/HOL looks as follows:
  - refl: "t = (t::'a)"
  - subst: "s = t ⟹ P s ⟹ P t"
  - ext: "(⋀x::'a. (f x ::'b) = g x) ⟹(λx. f x) = (λx. g x)"
  - the_eq_trivial: "(THE x. x = a) = (a::'a)"
  - impI:"(P ⟹ Q) ⟹ P⟶ Q"
  - mp: "P ⟶ Q ⟹ P ⟹ Q"
  - iff: "(P ⟶ Q) ⟶ (Q ⟶ P) ⟶ (P = Q)
  - True_or_False: "(P = True) ∨ (P = False)"

How to built theories in a logically safe manner ?

- where:
  - True is an abbreviation for $((\lambda x::bool.\ x) = (\lambda x.\ x))$
  - All(P) for $(P = (\lambda x.\ True))$
  - False for $(\forall P.\ P)$
  - Not P for $P \longrightarrow False$
  - and for $\forall R.\ (P \longrightarrow Q \longrightarrow R) \longrightarrow R$
  - or for $\forall R.\ (P \longrightarrow R) \longrightarrow (Q \longrightarrow R) \longrightarrow R$

How to built theories in a logically safe manner ?

- It is straight-forward to give Interpretation functions $I_{standard}$, I_{henkin} , $I_{construct}$ for HOL terms and formulas in ZFC

- (Meta) Theorem: Consistency relative to ZFC

  $I_{standard}$: τ => V and $I_{standard}$: T => V build a Model for Core-HOL, i.e. they satisfy all axioms for all interpretation of the free variables they contain.

- (Meta) Theorem: Incompleteness

  This model is incomplete for Core-HOL, i.e. there are always true terms for which this fact can not be derived.

How to built theories in a logically safe manner ?

- Is HOL correctly implemented in Isabelle ?
  - Isabelle as a system clearly contains bugs; but that does not mean that logical inferences produce false results
  - Isabelle has a kernel architecture it is a member of the LCF-style systems that protects „theorems“, i.e. triples of the form:

$$\Gamma \vdash_{\Theta} \varphi$$

    by a fairly small abstract data-type.

  - Isabelle can generate proof-objects which can be checked outside Isabelle, in principle by any other HOL prover.
  - It is heavily tested and used for a long time.

How to built theories in a logically safe manner ?

- Are Extensions of HOL, so for example, the library „Main“, logically safe ?

  - not necessarily, adding arbitrary axioms by the axiomatization command ruins consistency easily.
  - some proof-methods are not based on the kernel (sorry, self-built oracles, eval (code-generator))
  - However, Isabelle encourages to use specification constructs which are (in some cases even formally) shown to be conservative.

Isabelle Specification Constructs

- Constant Definitions:

```
definition f::"<τ>"
  where <name> : "f x₁ … xₙ = <t>"
```

example: definition C::"bool ⇒ bool"

where "C x = x"

- Type Definitions:

```
typedef ('α₁..'αₙ) κ =
  "<set-expr>" <proof>
```

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

Specification Commands

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

  (Σ, A) "$\in$" Θ

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

$(\Sigma \oplus f::\tau\ ,\ A \oplus$ "f $x_1 \ldots x_n$ = expr"$)$ "$\in$" Θ'

  – Side-Conditions
    - constant symbol f must be fresh
    - f must not be contained in "expr"
    - (all type-variables orrurring in expr must occur in τ)

Isabelle Specification Constructs

- Type definition:

(Σ, A) "$\in$" Θ

typedef $('\alpha_1..'\alpha_n)\ \kappa$ =

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

$$(\Sigma \oplus ('\alpha_1..'\alpha_n)\ \kappa \oplus \mathrm{Abs\_\kappa}::('\alpha_1..'\alpha_n)\tau \Rightarrow ('\alpha_1..'\alpha_n)\kappa$$
$$\oplus\ \mathrm{Rep\_\kappa}::('\alpha_1..'\alpha_n)\kappa \Rightarrow ('\alpha_1..'\alpha_n)\tau$$

$$A \oplus \{\mathrm{Rep\_\kappa\_inverse} \mapsto \mathrm{Abs\_\kappa}\ (\mathrm{Rep\_\kappa}\ x) = x\}$$

$$\oplus \{\mathrm{Rep\_\kappa\_inject} \mapsto (\mathrm{Rep\_\kappa}\ x = \mathrm{Rep\_\kappa}\ y) = (x = y)\}$$

$$\oplus \{\mathrm{Rep\_\kappa} \mapsto \mathrm{Rep\_\kappa}\ x \in \{x.\ \mathrm{expr}\ x\}) \text{"}\in\text{"}\ \Theta'$$

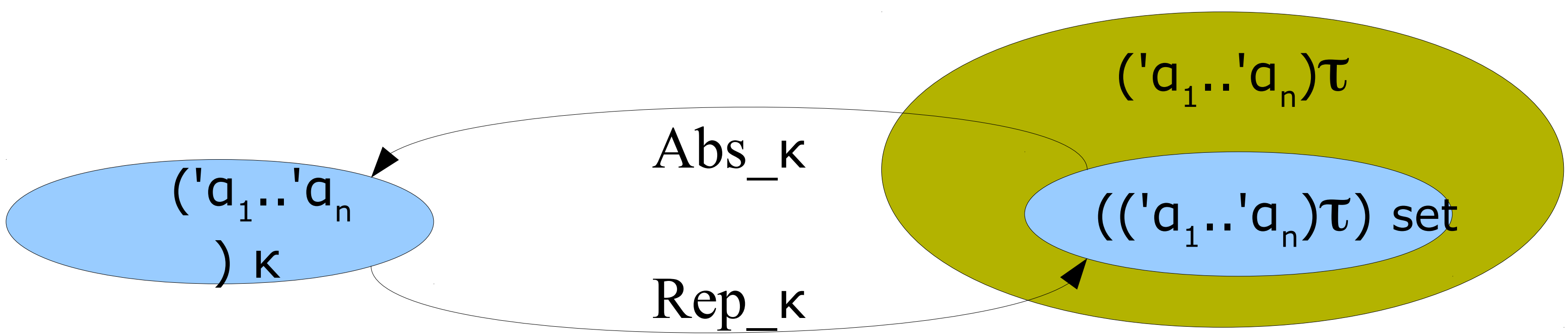
- where the type-constructor κ is "fresh" in Θ
- expr is closed
- <expr:: $('\alpha_1..'\alpha_n)\tau$ set> is non-empty (to be proven by a witness)

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) "∈" Θ

typedef $('\alpha_1..'\alpha_n)\ \kappa$ =

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

$$(\Sigma \oplus ('\alpha_1..'\alpha_n)\ \kappa \oplus \mathit{Abs\_\kappa}::('\alpha_1..'\alpha_n)\tau \Rightarrow ('\alpha_1..'\alpha_n)\kappa$$
$$\oplus\ \mathit{Rep\_\kappa}::('\alpha_1..'\alpha_n)\kappa \Rightarrow ('\alpha_1..'\alpha_n)\tau$$

$$A \oplus \{\mathit{Rep\_\kappa\_inverse} \mapsto \mathit{Abs\_\kappa}\ (\mathit{Rep\_\kappa}\ x) = x\}$$

$$\oplus \{\mathit{Rep\_\kappa\_inject} \mapsto (\mathit{Rep\_\kappa}\ x = \mathit{Rep\_\kappa}\ y) = (x = y)\}$$

$$\oplus \{\mathit{Rep\_\kappa} \mapsto \mathit{Rep\_\kappa}\ x \in \{x.\ \mathit{expr}\ x\})\ \text{"}\in\text{"}\ \Theta'$$

- 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)

Semantics of a „Type Definition“

- Major example: Typed sets can be built following
  this scheme. The trick is to identify α set with characteristic functions $\alpha \Rightarrow$ bool.

- In Isabelle/HOL, α set is introduced via an equivalen axiom scheme; the type-definition uses already implicitely the α set isomorphism to $\alpha \Rightarrow$ bool.

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)
```

Isabelle Specification Constructs

- Major example:

  Typed sets.

```
typedef ('a) set  = "prod :: ('a ⇒  'b ⇒  bool) set"
                                          unfolding prod_def by auto

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

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 : $x\ (\!|\ tag_1 := x_n\ |\!)$

Specification Mechanism Commands

- Inductively Defined Sets:

```
inductive_set     <c> :: " τ set"
              where  <thmname> : "<φ>"
                     | ...
                     | <thmname> = <φ>
```

```
inductive_set     <c> :: "  τ ⇒ τ} set" for A::τ
  where  <thmname> : "<φ>"
               | ...
               | <thmname> = <φ>
```

example:

```
inductive_set    Even :: "int set"
           where   null: "0 ∈ Even"
                 | plus:"x ∈ Even ⟹ x+2 ∈ Even"
                 | min :"x ∈ Even ⟹ x-2 ∈ Even"
```

Specification Mechanism Commands

- These are not buit-in constructs, rather they are based on a series of definitions and typedefs.
- The machinery behind is based on a fixed-point combinator on sets :

$$\text{lfp} :: \text{“}(\text{'}\alpha\ \text{set} \Rightarrow \text{'}\alpha\ \text{set}) \Rightarrow \text{'}\alpha\ \text{set”}$$

which can be conservatively defined by:

$$\text{"lfp } f = \bigcap \{u.\ f\ u \subseteq u\}\text{"}$$

and which enjoys a constrained fixed-point property:

$$\text{mono } f \Longrightarrow \text{lfp } f = f\ (\text{lfp } f)$$

Specification Mechanism Commands

- Example : Even (see before)
  - the set Even is conservatively defined by:

$$\text{Even} = \text{lfp}\ (\lambda\, X.\ \{0\} \cup (\lambda\, x.\ x + 2)\ `\ X \cup (\lambda\, x.\ x - 2)\ `\ X)$$

  - from which the properties:

null: "$0 \in$ Even"
plus:"$x \in$ Even $\Longrightarrow$ $x+2 \in$ Even"
min :"$x \in$ Even $\Longrightarrow$ $x-2 \in$ Even"

can be derived automatically (Note that Isabelle/HOL Version 2016 proceeds differently for technical reasons)

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

- 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"
```

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