A Compact Overview of Isabelle/HOL

Tobias Nipkow
Institut für Informatik, TU München
http://www.in.tum.de/~nipkow/

This document is a very compact introduction to the proof assistant Isabelle/HOL and is based directly on the published tutorial [1] where full explanations and a wealth of additional material can be found.

While reading this document it is recommended to single-step through the corresponding theories using Isabelle/HOL to follow the proofs.

1 Functional Programming/Modelling

1.1 An Introductory Theory

```
theory FP0 = PreList:

datatype 'a list = Nil                          ("[]")
                 | Cons 'a "'a list"            (infixr "#" 65)

consts app :: "'a list ⇒ 'a list ⇒ 'a list"   (infixr "@" 65)
       rev :: "'a list ⇒ 'a list"

primrec
"[] @ ys       = ys"
"(x # xs) @ ys = x # (xs @ ys)"

primrec
"rev []        = []"
"rev (x # xs)  = (rev xs) @ (x # [])"

theorem rev_rev [simp]: "rev(rev xs) = xs"

end
```

Exercise 1.1 Define a datatype of binary trees and a function `mirror` that mirrors a binary tree by swapping subtrees recursively. Prove `mirror(mirror t) = t`.

Define a function `flatten` that flattens a tree into a list by traversing it in infix order. Prove `flatten(mirror t) = rev(flatten t)`.

1.2 More Constructs

```
lemma "if xs = ys
       then rev xs = rev ys
       else rev xs ≠ rev ys"
by auto

lemma "case xs of
         []   ⇒ tl xs = xs
       | y#ys ⇒ tl xs ≠ xs"
apply(case_tac xs)
by auto
```

1.3 More Types

1.3.1 Natural Numbers

```
consts sum :: "nat ⇒ nat"
primrec "sum 0 = 0"
        "sum (Suc n) = Suc n + sum n"

lemma "sum n + sum n = n*(Suc n)"
apply(induct_tac n)
apply(auto)
done
```

Some examples of linear arithmetic:

```
lemma "⟦ ¬ m < n; m < n+(1::int) ⟧ ⟹ m = n"
by(auto)

lemma "min i (max j k) = max (min k i) (min i (j::nat))"
by(arith)
```

Not proved automatically because it involves multiplication:

```
lemma "n*n = n ⟹ n=0 ∨ n=(1::int)"
```

1.3.2 Pairs

```
lemma "fst(x,y) = snd(z,x)"
by auto
```

1.4 Definitions

```
consts xor :: "bool ⇒ bool ⇒ bool"
defs xor_def: "xor x y ≡ x ∧ ¬y ∨ ¬x ∧ y"

constdefs nand :: "bool ⇒ bool ⇒ bool"
         "nand x y ≡ ¬(x ∧ y)"
```

lemma *"¬ xor x x"*
apply *(unfold xor_def)*
by *auto*

1.5 Simplification

1.5.1 Simplification Rules

lemma *fst_conv[simp]: "fst(x,y) = x"*
by *auto*

Setting and resetting the *simp* attribute:

declare *fst_conv[simp]*
declare *fst_conv[simp del]*

1.5.2 The Simplification Method

lemma *"x*(y+1) = y*(x+1::nat)"*
apply *simp*

1.5.3 Adding and Deleting Simplification Rules

lemma *"∀x::nat. x*(y+z) = r"*
apply *(simp add: add_mult_distrib2)*

lemma *"rev(rev(xs @ [])) = xs"*
apply *(simp del: rev_rev_ident)*

1.5.4 Rewriting with Definitions

lemma *"xor A (¬A)"*
apply *(simp only: xor_def)*
apply *simp*
done

1.5.5 Conditional Equations

A pre-proved simplification rule: *xs ≠ [] ⟹ hd xs # tl xs = xs*

lemma *"hd(xs @ [x]) # tl(xs @ [x]) = xs @ [x]"*
by *simp*

1.5.6 Automatic Case Splits

lemma *"∀xs. if xs = [] then A else B"*
apply *simp*

Case-expressions are only split on demand.

1.5.7 Arithmetic

Only simple arithmetic:

```
lemma "⟦ ¬ m < n; m < n+(1::nat) ⟧ ⟹ m = n"
by simp
```

Complex goals need `arith`-method.

1.6 Case Study: Compiling Expressions

1.6.1 Expressions

```
types 'v binop = "'v ⇒ 'v ⇒ 'v"

datatype ('a,'v)expr = Cex 'v
                     | Vex 'a
                     | Bex "'v binop"  "('a,'v)expr"  "('a,'v)expr"

consts value :: "('a,'v)expr ⇒ ('a ⇒ 'v) ⇒ 'v"
primrec
"value (Cex v) env = v"
"value (Vex a) env = env a"
"value (Bex f e1 e2) env = f (value e1 env) (value e2 env)"
```

1.6.2 The Stack Machine

```
datatype ('a,'v) instr = Const 'v
                       | Load 'a
                       | Apply "'v binop"

consts exec :: "('a,'v)instr list ⇒ ('a⇒'v) ⇒ 'v list ⇒ 'v list"
primrec
"exec [] s vs = vs"
"exec (i#is) s vs = (case i of
    Const v  ⇒ exec is s (v#vs)
  | Load a   ⇒ exec is s ((s a)#vs)
  | Apply f  ⇒ exec is s ((f (hd vs) (hd(tl vs)))#(tl(tl vs))))"
```

1.6.3 The Compiler

```
consts comp :: "('a,'v)expr ⇒ ('a,'v)instr list"
primrec
"comp (Cex v)       = [Const v]"
"comp (Vex a)       = [Load a]"
"comp (Bex f e1 e2) = (comp e2) @ (comp e1) @ [Apply f]"

theorem "exec (comp e) s [] = [value e s]"
```

1.7 Advanced Datatypes

1.7.1 Mutual Recursion

```
datatype 'a aexp = IF   "'a bexp" "'a aexp" "'a aexp"
                 | Sum  "'a aexp" "'a aexp"
                 | Var 'a
                 | Num nat
and      'a bexp = Less "'a aexp" "'a aexp"
                 | And  "'a bexp" "'a bexp"
                 | Neg  "'a bexp"


consts  evala :: "'a aexp ⇒ ('a ⇒ nat) ⇒ nat"
        evalb :: "'a bexp ⇒ ('a ⇒ nat) ⇒ bool"

primrec
  "evala (IF b a1 a2) env =
     (if evalb b env then evala a1 env else evala a2 env)"
  "evala (Sum a1 a2) env = evala a1 env + evala a2 env"
  "evala (Var v) env = env v"
  "evala (Num n) env = n"

  "evalb (Less a1 a2) env = (evala a1 env < evala a2 env)"
  "evalb (And b1 b2) env = (evalb b1 env ∧ evalb b2 env)"
  "evalb (Neg b) env = (¬ evalb b env)"

consts substa :: "('a ⇒ 'b aexp) ⇒ 'a aexp ⇒ 'b aexp"
       substb :: "('a ⇒ 'b aexp) ⇒ 'a bexp ⇒ 'b bexp"

primrec
  "substa s (IF b a1 a2) =
     IF (substb s b) (substa s a1) (substa s a2)"
  "substa s (Sum a1 a2) = Sum (substa s a1) (substa s a2)"
  "substa s (Var v) = s v"
  "substa s (Num n) = Num n"

  "substb s (Less a1 a2) = Less (substa s a1) (substa s a2)"
  "substb s (And b1 b2) = And (substb s b1) (substb s b2)"
  "substb s (Neg b) = Neg (substb s b)"

lemma substitution_lemma:
 "evala (substa s a) env = evala a (λx. evala (s x) env) ∧
  evalb (substb s b) env = evalb b (λx. evala (s x) env)"
apply(induct_tac a and b)
by simp_all
```

1.7.2 Nested Recursion

```
datatype tree = C "tree list"
```

Some trees:

```
term "C []"
term "C [C [C []], C []]"

consts
mirror :: "tree ⇒ tree"
mirrors:: "tree list ⇒ tree list"

primrec
  "mirror(C ts) = C(mirrors ts)"

  "mirrors [] = []"
  "mirrors (t # ts) = mirrors ts @ [mirror t]"

lemma "mirror(mirror t) = t ∧ mirrors(mirrors ts) = ts"
apply(induct_tac t and ts)
apply simp_all
```

Exercise 1.2 Complete the above proof.

1.7.3 Datatypes Involving Functions

```
datatype ('a,'i)bigtree = Tip | Br 'a "'i ⇒ ('a,'i)bigtree"
```

A big tree:

```
term "Br 0 (λi. Br i (λn. Tip))"

consts map_bt :: "('a ⇒ 'b) ⇒ ('a,'i)bigtree ⇒ ('b,'i)bigtree"
primrec
"map_bt f Tip      = Tip"
"map_bt f (Br a F) = Br (f a) (λi. map_bt f (F i))"

lemma "map_bt (g o f) T = map_bt g (map_bt f T)"
apply(induct_tac T, rename_tac[2] F)
apply simp_all
done
```

This is *not* allowed:

```
datatype lambda = C "lambda => lambda"
```

Exercise 1.3 Define a datatype of ordinals and the ordinal Γ_0.

1.8 Wellfounded Recursion

1.8.1 Examples

```
consts fib :: "nat ⇒ nat"
recdef fib "measure(λn. n)"
  "fib 0 = 0"
  "fib (Suc 0) = 1"
  "fib (Suc(Suc x)) = fib x + fib (Suc x)"

consts sep :: "'a × 'a list ⇒ 'a list"
recdef sep "measure (λ(a,xs). length xs)"
  "sep(a, [])     = []"
  "sep(a, [x])    = [x]"
  "sep(a, x#y#zs) = x # a # sep(a,y#zs)"

consts last :: "'a list ⇒ 'a"
recdef last "measure (λxs. length xs)"
  "last [x]      = x"
  "last (x#y#zs) = last (y#zs)"

consts sep1 :: "'a × 'a list ⇒ 'a list"
recdef sep1 "measure (λ(a,xs). length xs)"
  "sep1(a, x#y#zs) = x # a # sep1(a,y#zs)"
  "sep1(a, xs)     = xs"
```

This is what the rules for *sep1* are turned into:

```
sep1 (a, x # y # zs) = x # a # sep1 (a, y # zs)
sep1 (a, []) = []
sep1 (a, [w]) = [w]
```

Pattern matching is also useful for nonrecursive functions:

```
consts swap12 :: "'a list ⇒ 'a list"
recdef swap12 "{}"
  "swap12 (x#y#zs) = y#x#zs"
  "swap12 zs       = zs"
```

1.8.2 Beyond Measure

The lexicographic product of two relations:

```
consts ack :: "nat×nat ⇒ nat"
recdef ack "measure(λm. m) <*lex*> measure(λn. n)"
  "ack(0,n)         = Suc n"
  "ack(Suc m,0)     = ack(m, 1)"
  "ack(Suc m,Suc n) = ack(m,ack(Suc m,n))"
```

1.8.3 Induction

Every recursive definition provides an induction theorem, for example *sep.induct*:

```
⟦⋀a. P a []; ⋀a x. P a [x];
   ⋀a x y zs. P a (y # zs) ⟹ P a (x # y # zs)⟧
⟹ P u v
```

```
lemma "map f (sep(x,xs)) = sep(f x, map f xs)"
apply(induct_tac x xs rule: sep.induct)
apply simp_all
done

lemma ack_incr2: "n < ack(m,n)"
apply(induct_tac m n rule: ack.induct)
apply simp_all
done
```

1.8.4 Recursion Over Nested Datatypes

```
datatype tree = C "tree list"

lemma termi_lem: "t ∈ set ts ⟶ size t < Suc(tree_list_size ts)"
by(induct_tac ts, auto)

consts mirror :: "tree ⇒ tree"
recdef mirror "measure size"
 "mirror(C ts) = C(rev(map mirror ts))"
(hints recdef_simp: termi_lem)

lemma "mirror(mirror t) = t"
apply(induct_tac t rule: mirror.induct)
apply(simp add: rev_map sym[OF map_compose] cong: map_cong)
done
```

Figure out how that proof worked!

Exercise 1.4 Define a function for merging two ordered lists (of natural numbers) and show that if the two input lists are ordered, so is the output.

2 The Rules of the Game

2.1 Introduction Rules

Introduction rules for propositional logic:

impI	$(P \Longrightarrow Q) \Longrightarrow P \longrightarrow Q$
conjI	$[\![P;\ Q]\!] \Longrightarrow P \wedge Q$
disjI1	$P \Longrightarrow P \vee Q$
disjI2	$Q \Longrightarrow P \vee Q$
iffI	$[\![P \Longrightarrow Q;\ Q \Longrightarrow P]\!] \Longrightarrow P = Q$

lemma *"A ⟹ B ⟶ A ∧ (B ∧ A)"*
apply *(rule impI)*
apply *(rule conjI)*
 apply *assumption*
apply *(rule conjI)*
 apply *assumption*
apply *assumption*
done

2.2 Elimination Rules

Elimination rules for propositional logic:

impE	$[\![P \longrightarrow Q;\ P;\ Q \Longrightarrow R]\!] \Longrightarrow R$
mp	$[\![P \longrightarrow Q;\ P]\!] \Longrightarrow Q$
conjE	$[\![P \wedge Q;\ [\![P;\ Q]\!] \Longrightarrow R]\!] \Longrightarrow R$
disjE	$[\![P \vee Q;\ P \Longrightarrow R;\ Q \Longrightarrow R]\!] \Longrightarrow R$

lemma *disj_swap: "P ∨ Q ⟹ Q ∨ P"*
apply *(erule disjE)*
 apply *(rule disjI2)*
 apply *assumption*
apply *(rule disjI1)*
apply *assumption*
done

2.3 Negation

notI	$(P \Longrightarrow False) \Longrightarrow \neg P$
notE	$[\![\neg P;\ P]\!] \Longrightarrow R$
ccontr	$(\neg P \Longrightarrow False) \Longrightarrow P$

lemma *"¬¬P ⟹ P"*
apply *(rule ccontr)*
apply *(erule notE)*
apply *assumption*
done

A simple exercise:

lemma *"¬P ∨ ¬Q ⟹ ¬(P ∧ Q)"*

2.4 Quantifiers

Quantifier rules:

allI	$(\bigwedge x.\ P\ x) \Longrightarrow \forall x.\ P\ x$
exI	$P\ x \Longrightarrow \exists x.\ P\ x$
allE	$[\![\forall x.\ P\ x;\ P\ x \Longrightarrow R]\!] \Longrightarrow R$
exE	$[\![\exists x.\ P\ x;\ \bigwedge x.\ P\ x \Longrightarrow Q]\!] \Longrightarrow Q$
spec	$\forall x.\ P\ x \Longrightarrow P\ x$

Another classic exercise:

```
lemma "∃x.∀y. P x y ⟹ ∀y.∃x. P x y"
```

2.5 Instantiation

```
lemma "∃xs. xs @ xs = xs"
apply(rule_tac x = "[]" in exI)
by simp
```

```
lemma "∀xs. f(xs @ xs) = xs ⟹ f [] = []"
apply(erule_tac x = "[]" in allE)
by simp
```

2.6 Automation

```
lemma "(∀x. honest(x) ∧ industrious(x) ⟶ healthy(x)) ∧
       ¬ (∃x. grocer(x) ∧ healthy(x)) ∧
       (∀x. industrious(x) ∧ grocer(x) ⟶ honest(x)) ∧
       (∀x. cyclist(x) ⟶ industrious(x)) ∧
       (∀x. ¬healthy(x) ∧ cyclist(x) ⟶ ¬honest(x))
       ⟶ (∀x. grocer(x) ⟶ ¬cyclist(x))"
by blast
```

```
lemma "(⋃i∈I. A(i)) ∩ (⋃j∈J. B(j)) =
       (⋃i∈I. ⋃j∈J. A(i) ∩ B(j))"
by fast
```

```
lemma "∃x.∀y. P x y ⟹ ∀y.∃x. P x y"
apply(clarify)
by blast
```

2.7 Forward Proof

Instantiation of unknowns (in left-to-right order):

append_self_conv	*(?xs @ ?ys = ?xs) = (?ys = [])*
append_self_conv [of _ "[]"]	*(?xs @ [] = ?xs) = ([] = [])*

Applying one theorem to another by unifying the premise of one theorem with the conclusion of another:

sym	*?s = ?t* $\Longrightarrow$ *?t = ?s*
sym [OF append_self_conv]	*(?ys1 = []) = (?xs1 @ ?ys1 = ?xs1)*
append_self_conv [THEN sym]	*(?ys1 = []) = (?xs1 @ ?ys1 = ?xs1)*

2.8 Further Useful Methods

```
lemma "n ≤ 1 ∧ n ≠ 0 ⟹ n^n = n"
apply(subgoal_tac "n=1")
 apply simp
apply arith
done
```

And a cute example:

```
lemma "⟦ 2 ∈ Q; sqrt 2 ∉ Q;
         ∀x y z. sqrt x * sqrt x = x ∧
                  x ^ 2 = x * x ∧
                 (x ^ y) ^ z = x ^ (y*z)
      ⟧ ⟹ ∃a b. a ∉ Q ∧ b ∉ Q ∧ a ^ b ∈ Q"
```

3 Sets, Functions and Relations

3.1 Set Notation

A ∪ *B*	*A* ∩ *B*	*A - B*
a ∈ *A*	*b* ∉ *A*	
{a, b}	*{x. P x}*	
⋃*M*	⋃*a* ∈ *A. F a*	

3.2 Some Functions

```
id ≡ λx. x
f ∘ g ≡ λx. f (g x)
f ` A ≡ {y. ∃x∈A. y = f x}
f -` B ≡ {x. f x ∈ B}
```

3.3 Some Relations

```
Id ≡ {p. ∃x. p = (x, x)}
r⁻¹ ≡ {(y, x). (x, y) ∈ r}
r `` s ≡ {y. ∃x∈s. (x, y) ∈ r}
(a, a) ∈ r*
⟦(a, b) ∈ r*; (b, c) ∈ r⟧ ⟹ (a, c) ∈ r*
r⁺ ≡ r O r*
```

3.4 Wellfoundedness

$wf\ r \equiv \forall P.\ (\forall x.\ (\forall y.\ (y,\ x) \in r \longrightarrow P\ y) \longrightarrow P\ x) \longrightarrow (\forall x.\ P\ x)$

$wf\ r = (\neg\ (\exists f.\ \forall i.\ (f\ (Suc\ i),\ f\ i) \in r))$

3.5 Fixed Point Operators

$lfp\ f \equiv \bigcap\{u.\ f\ u \subseteq u\}$

$mono\ f \Longrightarrow lfp\ f = f\ (lfp\ f)$

$[\![a \in lfp\ f;\ mono\ f;\ \bigwedge x.\ x \in f\ (lfp\ f \cap Collect\ P) \Longrightarrow P\ x]\!] \Longrightarrow P\ a$

3.6 Case Study: Verified Model Checking

typedecl *state*

consts M :: "(state $\times$ state)set"

typedecl *atom*

consts L :: "state $\Rightarrow$ atom set"

datatype *formula* = *Atom atom*
| *Neg formula*
| *And formula formula*
| *AX formula*
| *EF formula*

consts *valid* :: "state $\Rightarrow$ formula $\Rightarrow$ bool" ("(_ $\models$ _)" [80,80] 80)

primrec
"s $\models$ Atom a = (a $\in$ L s)"
"s $\models$ Neg f = ($\neg$(s $\models$ f))"
"s $\models$ And f g = (s $\models$ f $\wedge$ s $\models$ g)"
"s $\models$ AX f = ($\forall$t. (s,t) $\in$ M $\longrightarrow$ t $\models$ f)"
"s $\models$ EF f = ($\exists$t. (s,t) $\in$ M^* $\wedge$ t $\models$ f)"

consts *mc* :: "formula $\Rightarrow$ state set"
primrec
"mc(Atom a) = {s. a $\in$ L s}"
"mc(Neg f) = -mc f"
"mc(And f g) = mc f $\cap$ mc g"
"mc(AX f) = {s. $\forall$t. (s,t) $\in$ M $\longrightarrow$ t $\in$ mc f}"
"mc(EF f) = lfp(λT. mc f $\cup$ (M^{-1} `` T))"

lemma *mono_ef*: "mono(λT. A $\cup$ (M^{-1} `` T))"

```
apply(rule monoI)
apply blast
done

lemma EF_lemma:
  "lfp(λT. A ∪ (M⁻¹ `` T)) = {s. ∃t. (s,t) ∈ M* ∧ t ∈ A}"
apply(rule equalityI)
 thm lfp_lowerbound
 apply(rule lfp_lowerbound)
 apply(blast intro: rtrancl_trans)
apply(rule subsetI)
apply clarsimp
apply(erule converse_rtrancl_induct)
thm lfp_unfold[OF mono_ef]
 apply(subst lfp_unfold[OF mono_ef])
 apply(blast)
apply(subst lfp_unfold[OF mono_ef])
apply(blast)
done

theorem "mc f = {s. s ⊨ f}"
apply(induct_tac f)
apply(auto simp add: EF_lemma)
done
```

Exercise 3.1 *AX* has a dual operator *EN*[1] ("there exists a next state such that") with the intended semantics

`s ⊨ EN f = (∃t. (s, t) ∈ M ∧ t ⊨ f)`

Fortunately, *EN f* can already be expressed as a PDL formula. How?

Show that the semantics for *EF* satisfies the following recursion equation:

`s ⊨ EF f = (s ⊨ f ∨ s ⊨ EN (EF f))`

4 Inductive Definitions

4.1 Even Numbers

4.1.1 The Definition

```
consts even :: "nat set"
inductive even
```

[1]We cannot use the customary *EX* as that is the ASCII-equivalent of ∃

```
intros
zero[intro!]: "0 ∈ even"
step[intro!]: "n ∈ even ⟹ Suc(Suc n) ∈ even"

lemma [simp,intro!]: "2 dvd n ⟹ 2 dvd Suc(Suc n)"
apply (unfold dvd_def)
apply clarify
apply (rule_tac x = "Suc k" in exI, simp)
done
```

4.1.2 Rule Induction

Rule induction for set *even*, *even.induct*:

```
⟦xa ∈ even; P 0; ⋀n. ⟦n ∈ even; P n⟧ ⟹ P (Suc (Suc n))⟧ ⟹ P xa
```

```
lemma even_imp_dvd: "n ∈ even ⟹ 2 dvd n"
apply (erule even.induct)
apply simp_all
done
```

4.1.3 Rule Inversion

```
inductive_cases Suc_Suc_case [elim!]: "Suc(Suc n) ∈ even"
```

```
⟦Suc (Suc n) ∈ even; n ∈ even ⟹ P⟧ ⟹ P
```

```
lemma "Suc(Suc n) ∈ even ⟹ n ∈ even"
by blast
```

4.2 Mutually Inductive Definitions

```
consts evn :: "nat set"
       odd :: "nat set"

inductive evn odd
intros
zero: "0 ∈ evn"
evnI: "n ∈ odd ⟹ Suc n ∈ evn"
oddI: "n ∈ evn ⟹ Suc n ∈ odd"

lemma "(m ∈ evn ⟶ 2 dvd m) ∧ (n ∈ odd ⟶ 2 dvd (Suc n))"
apply(rule evn_odd.induct)
by simp_all
```

4.3 The Reflexive Transitive Closure

```
consts rtc :: "('a × 'a)set ⇒ ('a × 'a)set"   ("_*" [1000] 999)
inductive "r*"
intros
refl[iff]:  "(x,x) ∈ r*"
step:       "⟦ (x,y) ∈ r; (y,z) ∈ r* ⟧ ⟹ (x,z) ∈ r*"

lemma [intro]: "(x,y) : r ⟹ (x,y) ∈ r*"
by(blast intro: rtc.step)

lemma rtc_trans: "⟦ (x,y) ∈ r*; (y,z) ∈ r* ⟧ ⟹ (x,z) ∈ r*"
apply(erule rtc.induct)
oops

lemma rtc_trans[rule_format]:
  "(x,y) ∈ r* ⟹ (y,z) ∈ r* ⟶ (x,z) ∈ r*"
apply(erule rtc.induct)
 apply(blast)
apply(blast intro: rtc.step)
done
```

Exercise 4.1 Show that the converse of `rtc.step` also holds:

`⟦(x, y) ∈ r*; (y, z) ∈ r⟧ ⟹ (x, z) ∈ r*`

4.4 The accessible part of a relation

```
consts  acc :: "('a × 'a) set ⇒ 'a set"
inductive "acc r"
intros
  "(∀y. (y,x) ∈ r ⟶ y ∈ acc r) ⟹ x ∈ acc r"

lemma "wf{(x,y). (x,y) ∈ r ∧ y ∈ acc r}"
thm wfI
apply(rule_tac A = "acc r" in wfI)
 apply (blast elim: acc.elims)
apply(simp(no_asm_use))
thm acc.induct
apply(erule acc.induct)
by blast

consts  accs :: "('a × 'a) set ⇒ 'a set"
inductive "accs r"
intros
 "r``{x} ∈ Pow(accs r) ⟹ x ∈ accs r"
```

monos *Pow_mono*

References

[1] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2002. http://www.in.tum.de/~nipkow/LNCS2283/.