$(\forall\iota\sigma\alpha\Theta\exists\iota\sigma\alpha\Pi)$

Notes

i12prI2.thy

Gottfried Barrow

August 6, 2012

Contents

Theory Header

```
theory i12prI2
imports Complex_Main
        "../../pi/I"   (*Declare, print, sledge, nitP cmds. Not really needed.*)
begin
```

2 Programming and Proving [3.1.a]

2.1 Basics [3.2.b]

2.1.1 Predefined Ops and Funcprog Constructs [3.4.a]

(σ | Syntax) 2.1.2. "Op * X Y" Is The Func For Predefined Binary Infix Functions

```
lemma testOp: "op ∧ True True"
  by auto
```

2.1.3 Lambda-Abstractions, Formulae, Equality, Quantifiers [4.1.a]

(ν | Note) 2.1.4 (*abstractions*). Terms Can Contain Lambda Abstractions.

```
lemma lambdaInTerm: "((λx. x) a) = a"
  by auto
```

2.1.5 Not, And, Or, Imp, Quantifiers [4.2.b]

(ω | Operator) 2.1.6. not, or, and, implication

```
lemma notOp: "¬False"
  by auto
lemma andOpα: "op ∧ True True"
  by auto
lemma andOp2: "True ∧ True"
  by auto
lemma orOp: "op ∨ True False"
  by auto
lemma impOp: "op ⟶ False True"
  by auto
lemma impOp2: "False ⟶ True"
  by auto
```

(ω | Operator) 2.1.7. Equality "Op =" Means Iff For Formulae

```
lemma equalityOp: "op = a a"
  by auto
lemma equalityIff: "(op = a a) = (op = b b)"
  by auto
```

(ω | Operator) 2.1.8. All, Exists

```
lemma forallOp: "∀x. x=x"
  by auto
lemma existsOp: "∃x. x=False"
  by auto
```

(ω | Operator) 2.1.9. Meta-logic: and, implication, equivalence

```
lemma metalogicAnd: "⋀x. x = x"
  by auto
lemma metalogicImp: "False ⟹ True"
  by auto
lemma metalogicEqual: "a ≡ a"
  by auto
```

2.1.10 Right Arrows Associate Right and Imp Shorthand [4.3$_{.c}$]

(σ | Syntax) 2.1.11. meta-implication shorthand

```
lemma impShorthand1: "⟦False;True⟧ ⟹ True"
  by auto
lemma impLonghand: "False ⟹ True ⟹ True"
  by auto
lemma impShorthand4: "[|True;True|] ==> True"
  by auto
```

2.2 Types bool, nat and list [5.3$_{.a}$]

2.2.1 Type bool [5.3$_{.b}$]

```
(*Keyword "datatype" defines an inductive datatype in HOL [pi12rRef.{213}]. It
  requires Datatype.thy.*)
datatype boolTest = TrueTest | FalseTest
value "TrueTest" value "FalseTest"

(*However, though the tutorial shows that bool is defined by datatype, it's
  actually defined with "typedecl bool" in HOL.thy.*)
typedecl boolTest2
(*judgment
  Trueprop2      :: "boolTest2 => prop"*)
  (*But I get this error unless commented out: "Attempt to redeclare object-logic
    judgment". That's because there can only be one judgement per theory
    [pi12rRef.{195}].*)
consts
  TrueTest2          :: bool
  FalseTest2         :: bool
```

2.2.2 How conj could be defined [5.4$_{.c}$]

```
(*The keyword "fun" requires FunDef.thy.*)
fun conjTest :: "bool => bool => bool" where
  "conjTest True True = True" |
  "conjTest _ _       = False"
value "conjTest True True"
value "conj True True" (*using the normal conjuction*)
```

2.2.3 TYPE NAT [6.1$_{.a}$]

```
value "0::nat"
value "Suc(0::nat)"
```

2.2.4 NAT ARITHMETIC AND COMPARISON FUNCTIONS [6.1$_{.b}$]

```
value "0 ≤ Suc 0"
value "0 ≥ Suc 0"
value "Suc(Suc(Suc 0))"
value "Suc(Suc(Suc 0)) + Suc(Suc(Suc 0))"
value "Suc(Suc(Suc 0)) - Suc(Suc(Suc 0))"
value "Suc(Suc 0) - Suc(Suc(Suc 0))"
value "Suc 0 = 1"
value "Suc(Suc(Suc 0)) + Suc(Suc(Suc 0)) = 6"
value "Suc(Suc(Suc 0)) * Suc(Suc(Suc 0)) * Suc(Suc(Suc 0))
       =  3^3"
```

2.2.5 NAT ADD, A LEMMA AND PROOF [6.2$_{.c}$]

```
fun add :: "nat => nat => nat" where
  "add      0  n = n" |
  "add (Suc m) n = Suc(add m n)"
value "add 1 2::nat"
value "add m 0 = m"

theorem add: "add m 0 = m"
  (*JV PROOF:
    BASIS: Let m=0. Then 0+0=0, by the first condition of addT definition.
    INDUCTION STEP: Assume that m+0=m. Need to show that (Suc m) + 0 = Suc m.
    By definition of addT and m+0=m, (Suc m)+0 = Suc(m+0) = Suc m.
  *)
  apply(induction m)
  apply(auto)
  done

thm add

lemma stuffer1: "x + 0 = x"
  oops
lemma stuffer2: "x + (0::nat) = x"
  oops

value "0";
```

2.2.6 TYPE LIST [8.1.a]

```
datatype 'a list2 --"List type. Renamed to not clash with List.list."
  =Nil2
  |Cons2 'a "'a list2"
--"𝒳𝒫:"
  value "Cons (2::nat) (Nil::nat list)"
  value "Cons (2::nat) (Nil)"
  value "Cons d (Cons c (Cons a (Cons b (Nil))))"
  value "Cons x Nil"
  value "Nil = Nil"
  value "Cons x y"

fun app --"Append one list to another list [pg.8]"::
  "'a list ⇒ 'a list ⇒ 'a list" where
--"ℱ𝒰:"
  "app Nil ys = ys" |
  "app (Cons x xs) ys = Cons x (app xs ys)"
--"𝒳𝒫:"
  value "app (Cons x Nil) Nil"
  value "app (Cons x Nil) (Cons x Nil)"

fun rev --"Reverse a list."::
  "'a list ⇒ 'a list" where
--"ℱ𝒰:"
  "rev Nil = Nil" |
  "rev (Cons x xs) = app (rev xs) (Cons x Nil)"
--"𝒳𝒫:"
  value "Cons False Nil"
  value "rev(Cons True (Cons False Nil))"
  value "rev(Cons a (Cons b Nil))"
```

2.2.7 2.2.4 THE PROOF PROCESS [9.3.a]

```
theorem rev_rev --"Reversing a list twice gives the original list."[simp]:
  "rev(rev xs) = xs"
--"𝒫ℱ:"
  apply(induction xs)
  apply(auto)
  oops                    (*Oops. Need a lemma.*)

lemma rev_app [simp]:
  "rev(app xs ys) = app(rev ys)(rev xs)"
--"𝒫ℱ:"
  apply(induction xs)
  apply(auto) oops        (*Oops. Auto doesn't even get rid of step 1.*)

lemma app_Nil2 [simp]:
  "app xs Nil = xs"
--"𝒫ℱ:"
  apply(induction xs)
  by(auto)                (*Works. And it's added to simp.*)

lemma rev_app [simp]:
  "rev(app xs ys) = app(rev ys)(rev xs)"
```

```
--"𝒫ℱ:"
  apply(induction xs)
  apply(auto) oops         (*Oops. The lemma above in simp solves the base case.
                             But the inductive step has only been simplified to
                             a point where it needs associativity.*)

lemma app_assoc [simp]:
  "app (app xs ys) zs = app xs (app ys zs)"
--"𝒫ℱ:"
  apply(induction xs)
  by(auto)                 (*Works. Back to rev_app.*)

lemma rev_app [simp]:
  "rev(app xs ys) = app(rev ys)(rev xs)"
--"𝒫ℱ:"
  apply(induction xs)
  by(auto)                 (*Works. Back to rev_rev.*)

theorem rev_rev --"Reversing a list twice equals the original list."[simp]:
  "rev(rev xs) = xs"
--"𝒫ℱ:"
  apply(induction xs)
  by(auto)
```

2.2.8 2.2.5 PREDEFINED LISTS [12.2$_{.a}$]

2.2.9 Standard list operators [12.2$_{.b}$]

```

--"Standard list operators."
  value "List.list.Nil"        (* "[]" :: "'a List.list" *)
  value "a::nat List.list"     (* "a" :: "nat List.list" *)
  value "(2::nat) # []"        (* "[2]"::"nat List.list". Same as "Cons 2 []".*)
  value "List.Cons (2::nat) []"       (* Same as above. *)
  value "List.Cons x xs"              (* "x # xs"::"'a List.list" *)
  value "[a,b,c] = a # b # c # []"    (* "True" :: "bool" *)
  value "(append xs ys) = (xs @ ys)"  (* "True" :: "bool" *)
  value "append xs ys"                (* "xs @ ys" :: "'a List.list" *)
  value "xs @ ys"
  value "[1::nat,2] @ [3,4]"
  value "append xs ys = xs @ ys"

fun add1 ::
  "nat => nat" where
--"ℱ𝒰:"
  "add1 x = x + 1"
--"𝒳𝒫:"
  value "[1::nat,2]"             (* "[1, 2]" :: "nat List.list" *)
  value "length"
  value "length [1::nat,2]"      (* "2" :: "nat" *)
  value "map"
  value "map add1 [1::nat,2]"    (* "[2, 3]" :: "nat List.list" *)
```

2.3 Type and function definitions [$12.3_{.a}$]

2.3.1 General datatype definition [$12.3_{.b}$]

```

datatype ('a,'b)test2_3_1
  =Nil
  |Con "'a" "'b"
```

2.3.2 Structural induction for the general datatype [$13.1_{.a}$]

```

--"HOW TO PROVE INDUCTION."
(*For P, prove P(Nil). Then assume P(xs) and prove P(Cons x xs).*)
(*PG.13, To show P x for all x of type ('a1,...,'an)t:
  Assume: P(x_j) for all j where t_i,j = ('a_1,...,'a_n)t.*)

datatype 'a tree --"As an example, consider binary trees [pg.13]."
  =Tip
  |Node "'a tree" 'a  "'a tree"
--"𝒩ℰ:"
    --"i12tu.{17}: size is 1 plus the sum of all the args of type t. The size of
       the args of Tip is 0, so (size Tip) is 1."
--"𝒳𝒫:"
  value "size Tip"
  value "size (Node Tip y Tip)"                          (*size = 1*)
  value "size (Node (Node Tip y Tip) y (Node Tip y Tip))" (*size = 3*)

fun mirror --"A mirror function for datatype tree, [pg.13]."::
  "'a tree ⇒ 'a tree" where
--"ℱ𝒰:"
  "mirror Tip = Tip" |
  "mirror (Node l a r) = Node (mirror r) a (mirror l)"

lemma --"The following lemma illustrates induction:"
  "mirror(mirror t) = t"
--"𝒫ℱ:"
  apply(induction t)
  by(auto)
```

2.3.3 Definitions and Abbreviations [$14.2_{.a}$]

```

definition sq --"Non recursive functions are defined as in this example."::
  "nat ⇒ nat" where
--"𝒟ℱ:"
  "sq n = n * n"
--"𝒳𝒫:"
  value "sq n"
  value "sq 20"

abbreviation sq' --"Abbreviations are similar to definitions:"::
  "nat ⇒ nat" where "sq' n == n * n"
--"𝒳𝒫:"
  value "(n::nat) * n"
  value "sq' (n::nat)"
```

2.3.4 Recursive functions [$14.3_{.b}$]

```
fun div2 --"Functions defined with fun come with their own induction schema"::
  "nat ⇒ nat" where
--"ℱ𝒰:"
  "div2 0            = 0" |
  "div2 (Suc 0)      = Suc 0" |
  "div2 (Suc(Suc n)) = Suc(div2 n)"
--"𝒳𝒫:"
  (*The size of RHS arg of div2, n, is smaller than the RHS div2, Suc(Suc n).*)
  value "div2 1"
  value "div2 23"
  value "div2 ((n::nat)+n)"

lemma --"This customized induction rule can simplify inductive proofs."
  "div2(n+n) = n"
--"𝒫ℱ:"
  apply(induction n rule: div2.induct)
  by(auto)

lemma --"div2 using only apply(induction n)."
  "div2(n+n) = n"
--"𝒫ℱ:"
  apply(induction n)
  by(auto)
```

2.4 Induction heuristics [$16.1_{.a}$]

```
fun itrev --"A linear time version of rev"::
  "'a list ⇒ 'a list ⇒ 'a list" where
--"ℱ𝒰:"
  "itrev []     ys = ys" |
  "itrev (x#xs) ys = itrev xs (x#ys)" print_theorems
--"𝒳𝒫:"
  value "itrev [1::nat,2,3,4] []"
  value "itrev [] [1::nat,2,3,4]"
  value "itrev [1::nat,2,3,4] [8,9]"

lemma --"Only 1 variable, so the induction hypothesis is too weak."
  "itrev xs [] = rev xs"
  apply(induction xs)
  apply(auto)
  oops

lemma --"2 variables now, but only xs is suitable for induction."
  "itrev xs ys = rev xs @ ys"
  --"ℱ𝒰:"
    --"rev Nil = Nil"
    --"rev (Cons x xs) = app (rev xs) (Cons x Nil)"
  --"ℱ𝒰:"
    --"app Nil ys = ys"
    --"app (Cons x xs) ys = Cons x (app xs ys)"
  apply(induction xs, auto)
  oops
```

Theory End

end

[—i12prI2.thy—]

```
theory i12prI2
imports Complex_Main
        "../../pi/I"   (*Declare, print, sledge, nitP cmds. Not really needed.*)
begin
lemma testOp: "op ∧ True True"
  by auto
lemma lambdaInTerm: "((λx. x) a) = a"
  by auto
lemma notOp: "¬False"
  by auto
lemma andOp_α: "op ∧ True True"
  by auto
lemma andOp_2: "True ∧ True"
  by auto
lemma orOp: "op ∨ True False"
  by auto
lemma impOp: "op ⟶ False True"
  by auto
lemma impOp_2: "False ⟶ True"
  by auto
lemma equalityOp: "op = a a"
  by auto
lemma equalityIff: "(op = a a) = (op = b b)"
  by auto
lemma forallOp: "∀x. x=x"
  by auto
lemma existsOp: "∃x. x=False"
  by auto
lemma metalogicAnd: "⋀x. x = x"
  by auto
lemma metalogicImp: "False ⟹ True"
  by auto
lemma metalogicEqual: "a ≡ a"
  by auto
lemma impShorthand1: "⟦False;True⟧ ⟹ True"
  by auto
lemma impLonghand: "False ⟹ True ⟹ True"
  by auto
lemma impShorthand4: "[|True;True|] ==> True"
  by auto
(*Keyword "datatype" defines an inductive datatype in HOL [pi12rRef.{213}]. It
  requires Datatype.thy.*)
datatype boolTest = TrueTest | FalseTest
value "TrueTest" value "FalseTest"

(*However, though the tutorial shows that bool is defined by datatype, it's
  actually defined with "typedecl bool" in HOL.thy.*)
typedecl boolTest2
(*judgment
  Trueprop2      :: "boolTest2 => prop"*)
  (*But I get this error unless commented out: "Attempt to redeclare object-logic
    judgment". That's because there can only be one judgement per theory
    [pi12rRef.{195}].*)
consts
  TrueTest2          :: bool
  FalseTest2         :: bool
```

```
(*The keyword "fun" requires FunDef.thy.*)
fun conjTest :: "bool => bool => bool" where
  "conjTest True True = True" |
  "conjTest _ _       = False"
value "conjTest True True"
value "conj True True" (*using the normal conjuction*)
value "0::nat"
value "Suc(0::nat)"
value "0 ≤ Suc 0"
value "0 ≥ Suc 0"
value "Suc(Suc(Suc 0))"
value "Suc(Suc(Suc 0)) + Suc(Suc(Suc 0))"
value "Suc(Suc(Suc 0)) - Suc(Suc(Suc 0))"
value "Suc(Suc 0) - Suc(Suc(Suc 0))"
value "Suc 0 = 1"
value "Suc(Suc(Suc 0)) + Suc(Suc(Suc 0)) = 6"
value "Suc(Suc(Suc 0)) * Suc(Suc(Suc 0)) * Suc(Suc(Suc 0))
       =  3^3"
fun add :: "nat => nat => nat" where
  "add      0  n = n" |
  "add (Suc m) n = Suc(add m n)"
value "add 1 2::nat"
value "add m 0 = m"

theorem add: "add m 0 = m"
  (*JV PROOF:
    BASIS: Let m=0. Then 0+0=0, by the first condition of addT definition.
    INDUCTION STEP: Assume that m+0=m. Need to show that (Suc m) + 0 = Suc m.
    By definition of addT and m+0=m, (Suc m)+0 = Suc(m+0) = Suc m.
  *)
  apply(induction m)
  apply(auto)
  done

thm add

lemma stuffer1: "x + 0 = x"
  oops
lemma stuffer2: "x + (0::nat) = x"
  oops

value "0";

datatype 'a list2 --"List type. Renamed to not clash with List.list."
  =Nil2
  |Cons2 'a "'a list2"
--"𝒳𝒫:"
  value "Cons (2::nat) (Nil::nat list)"
  value "Cons (2::nat) (Nil)"
  value "Cons d (Cons c (Cons a (Cons b (Nil))))"
  value "Cons x Nil"
  value "Nil = Nil"
  value "Cons x y"

fun app --"Append one list to another list [pg.8]"::
  "'a list ⇒ 'a list ⇒ 'a list" where
--"ℱ𝒰:"
  "app Nil ys = ys" |
```

```
  "app (Cons x xs) ys = Cons x (app xs ys)"
--"𝒳𝒫:"
  value "app (Cons x Nil) Nil"
  value "app (Cons x Nil) (Cons x Nil)"

fun rev --"Reverse a list."::
  "'a list ⇒ 'a list" where
--"ℱ𝒰:"
  "rev Nil = Nil" |
  "rev (Cons x xs) = app (rev xs) (Cons x Nil)"
--"𝒳𝒫:"
  value "Cons False Nil"
  value "rev(Cons True (Cons False Nil))"
  value "rev(Cons a (Cons b Nil))"

theorem rev_rev --"Reversing a list twice gives the original list."[simp]:
  "rev(rev xs) = xs"
--"𝒫ℱ:"
  apply(induction xs)
  apply(auto)
  oops                      (*Oops. Need a lemma.*)

lemma rev_app [simp]:
  "rev(app xs ys) = app(rev ys)(rev xs)"
--"𝒫ℱ:"
  apply(induction xs)
  apply(auto) oops          (*Oops. Auto doesn't even get rid of step 1.*)

lemma app_Nil2 [simp]:
  "app xs Nil = xs"
--"𝒫ℱ:"
  apply(induction xs)
  by(auto)                  (*Works. And it's added to simp.*)

lemma rev_app [simp]:
  "rev(app xs ys) = app(rev ys)(rev xs)"
--"𝒫ℱ:"
  apply(induction xs)
  apply(auto) oops          (*Oops. The lemma above in simp solves the base case.
                              But the inductive step has only been simplified to
                              a point where it needs associativity.*)

lemma app_assoc [simp]:
  "app (app xs ys) zs = app xs (app ys zs)"
--"𝒫ℱ:"
  apply(induction xs)
  by(auto)                  (*Works. Back to rev_app.*)

lemma rev_app [simp]:
  "rev(app xs ys) = app(rev ys)(rev xs)"
--"𝒫ℱ:"
  apply(induction xs)
  by(auto)                  (*Works. Back to rev_rev.*)

theorem rev_rev --"Reversing a list twice equals the original list."[simp]:
  "rev(rev xs) = xs"
--"𝒫ℱ:"
  apply(induction xs)
```

```
  by(auto)

--"Standard list operators."
  value "List.list.Nil"        (* "[]" :: "'a List.list" *)
  value "a::nat List.list"     (* "a" :: "nat List.list" *)
  value "(2::nat) # []"        (* "[2]"::"nat List.list". Same as "Cons 2 []".*)
  value "List.Cons (2::nat) []"       (* Same as above. *)
  value "List.Cons x xs"              (* "x # xs"::"'a List.list" *)
  value "[a,b,c] = a # b # c # []"    (* "True" :: "bool" *)
  value "(append xs ys) = (xs @ ys)"  (* "True" :: "bool" *)
  value "append xs ys"                (* "xs @ ys" :: "'a List.list" *)
  value "xs @ ys"
  value "[1::nat,2] @ [3,4]"
  value "append xs ys = xs @ ys"

fun add1 ::
  "nat => nat" where
--"𝓕𝓤:"
  "add1 x = x + 1"
--"𝓧𝓟:"
  value "[1::nat,2]"              (* "[1, 2]" :: "nat List.list" *)
  value "length"
  value "length [1::nat,2]"       (* "2" :: "nat" *)
  value "map"
  value "map add1 [1::nat,2]"     (* "[2, 3]" :: "nat List.list" *)

datatype ('a,'b)test2_3_1
  =Nil
  |Con "'a" "'b"

--"HOW TO PROVE INDUCTION."
(*For P, prove P(Nil). Then assume P(xs) and prove P(Cons x xs).*)
(*PG.13, To show P x for all x of type ('a1,...,'an)t:
  Assume: P(x_j) for all j where t_i,j = ('a_1,...,'a_n)t.*)

datatype 'a tree --"As an example, consider binary trees [pg.13]."
  =Tip
  |Node "'a tree" 'a  "'a tree"
--"𝓝𝓔:"
    --"i12tu.{17}: size is 1 plus the sum of all the args of type t. The size of
       the args of Tip is 0, so (size Tip) is 1."
--"𝓧𝓟:"
  value "size Tip"
  value "size (Node Tip y Tip)"                           (*size = 1*)
  value "size (Node (Node Tip y Tip) y (Node Tip y Tip))" (*size = 3*)

fun mirror --"A mirror function for datatype tree, [pg.13]."::
  "'a tree ⇒ 'a tree" where
--"𝓕𝓤:"
  "mirror Tip = Tip" |
  "mirror (Node l a r) = Node (mirror r) a (mirror l)"

lemma --"The following lemma illustrates induction:"
  "mirror(mirror t) = t"
--"𝓟𝓕:"
  apply(induction t)
  by(auto)
```

```
definition sq --"Non recursive functions are defined as in this example."::
  "nat ⇒ nat" where
--"𝒟ℱ:"
  "sq n = n * n"
--"𝒳𝒫:"
  value "sq n"
  value "sq 20"

abbreviation sq' --"Abbreviations are similar to definitions:"::
  "nat ⇒ nat" where "sq' n == n * n"
--"𝒳𝒫:"
  value "(n::nat) * n"
  value "sq' (n::nat)"

fun div2 --"Functions defined with fun come with their own induction schema"::
  "nat ⇒ nat" where
--"ℱ𝒰:"
  "div2 0             = 0" |
  "div2 (Suc 0)       = Suc 0" |
  "div2 (Suc(Suc n)) = Suc(div2 n)"
--"𝒳𝒫:"
  (*The size of RHS arg of div2, n, is smaller than the RHS div2, Suc(Suc n).*)
  value "div2 1"
  value "div2 23"
  value "div2 ((n::nat)+n)"

lemma --"This customized induction rule can simplify inductive proofs."
  "div2(n+n) = n"
--"𝒫ℱ:"
  apply(induction n rule: div2.induct)
  by(auto)

lemma --"div2 using only apply(induction n)."
  "div2(n+n) = n"
--"𝒫ℱ:"
  apply(induction n)
  by(auto)
fun itrev --"A linear time version of rev"::
  "'a list ⇒ 'a list ⇒ 'a list" where
--"ℱ𝒰:"
  "itrev []     ys = ys" |
  "itrev (x#xs) ys = itrev xs (x#ys)" print_theorems
--"𝒳𝒫:"
  value "itrev [1::nat,2,3,4] []"
  value "itrev [] [1::nat,2,3,4]"
  value "itrev [1::nat,2,3,4] [8,9]"

lemma --"Only 1 variable, so the induction hypothesis is too weak."
  "itrev xs [] = rev xs"
  apply(induction xs)
  apply(auto)
  oops

lemma --"2 variables now, but only xs is suitable for induction."
  "itrev xs ys = rev xs @ ys"
  --"ℱ𝒰:"
    --"rev Nil = Nil"
    --"rev (Cons x xs) = app (rev xs) (Cons x Nil)"
```

```
  --"𝓕𝓤:"
    --"app Nil ys = ys"
    --"app (Cons x xs) ys = Cons x (app xs ys)"
  apply(induction xs, auto)
  oops



end
```

[—i12prI2.thy \<cmds>—]

```
theory i12prI2
imports Complex_Main
        "../../pi/I"   (*Declare, print, sledge, nitP cmds. Not really needed.*)
begin
lemma testOp: "op \<and> True True"
  by auto
lemma lambdaInTerm: "((\<lambda>x. x) a) = a"
  by auto
lemma notOp: "\<not>False"
  by auto
lemma andOp\<^isub>\<alpha>: "op \<and> True True"
  by auto
lemma andOp\<^isub>2: "True \<and> True"
  by auto
lemma orOp: "op \<or> True False"
  by auto
lemma impOp: "op \<longrightarrow> False True"
  by auto
lemma impOp\<^isub>2: "False \<longrightarrow> True"
  by auto
lemma equalityOp: "op = a a"
  by auto
lemma equalityIff: "(op = a a) = (op = b b)"
  by auto
lemma forallOp: "\<forall>x. x=x"
  by auto
lemma existsOp: "\<exists>x. x=False"
  by auto
lemma metalogicAnd: "\<And>x. x = x"
  by auto
lemma metalogicImp: "False \<Longrightarrow> True"
  by auto
lemma metalogicEqual: "a \<equiv> a"
  by auto
lemma impShorthand1: "\<lbrakk>False;True\<rbrakk> \<Longrightarrow> True"
  by auto
lemma impLonghand: "False \<Longrightarrow> True \<Longrightarrow> True"
  by auto
lemma impShorthand4: "[|True;True|] ==> True"
  by auto
(*Keyword "datatype" defines an inductive datatype in HOL [pi12rRef.{213}]. It
  requires Datatype.thy.*)
datatype boolTest = TrueTest | FalseTest
value "TrueTest" value "FalseTest"

(*However, though the tutorial shows that bool is defined by datatype, it's
  actually defined with "typedecl bool" in HOL.thy.*)
typedecl boolTest2
(*judgment
  Trueprop2      :: "boolTest2 => prop"*)
  (*But I get this error unless commented out: "Attempt to redeclare object-logic
    judgment". That's because there can only be one judgement per theory
    [pi12rRef.{195}].*)
consts
  TrueTest2          :: bool
  FalseTest2         :: bool
(*The keyword "fun" requires FunDef.thy.*)
fun conjTest :: "bool => bool => bool" where
  "conjTest True True = True" |
  "conjTest _ _       = False"
value "conjTest True True"
value "conj True True" (*using the normal conjuction*)
value "0::nat"
value "Suc(0::nat)"
value "0 \<le> Suc 0"
value "0 \<ge> Suc 0"
value "Suc(Suc(Suc 0))"
value "Suc(Suc(Suc 0)) + Suc(Suc(Suc 0))"
value "Suc(Suc(Suc 0)) - Suc(Suc(Suc 0))"
value "Suc(Suc 0) - Suc(Suc(Suc 0))"
value "Suc 0 = 1"
value "Suc(Suc(Suc 0)) + Suc(Suc(Suc 0)) = 6"
value "Suc(Suc(Suc 0)) * Suc(Suc(Suc 0)) * Suc(Suc(Suc 0))
        =  3^3"
fun add :: "nat => nat => nat" where
  "add      0  n = n" |
  "add (Suc m) n = Suc(add m n)"
value "add 1 2::nat"
value "add m 0 = m"

theorem add: "add m 0 = m"
  (*JV PROOF:
    BASIS: Let m=0. Then 0+0=0, by the first condition of addT definition.
    INDUCTION STEP: Assume that m+0=m. Need to show that (Suc m) + 0 = Suc m.
    By definition of addT and m+0=m, (Suc m)+0 = Suc(m+0) = Suc m.
  *)
  apply(induction m)
  apply(auto)
  done

thm add

lemma stuffer1: "x + 0 = x"
  oops
lemma stuffer2: "x + (0::nat) = x"
  oops
```

```

value "0";

datatype 'a list2 --"List type. Renamed to not clash with List.list."
  =Nil2
  |Cons2 'a "'a list2"
--"\<X>\<P>:"
  value "Cons (2::nat) (Nil::nat list)"
  value "Cons (2::nat) (Nil)"
  value "Cons d (Cons c (Cons a (Cons b (Nil))))"
  value "Cons x Nil"
  value "Nil = Nil"
  value "Cons x y"

fun app --"Append one list to another list [pg.8]"::
  "'a list \<Rightarrow> 'a list \<Rightarrow> 'a list" where
--"\<F>\<U>:"
  "app Nil ys = ys" |
  "app (Cons x xs) ys = Cons x (app xs ys)"
--"\<X>\<P>:"
  value "app (Cons x Nil) Nil"
  value "app (Cons x Nil) (Cons x Nil)"

fun rev --"Reverse a list."::
  "'a list \<Rightarrow> 'a list" where
--"\<F>\<U>:"
  "rev Nil = Nil" |
  "rev (Cons x xs) = app (rev xs) (Cons x Nil)"
--"\<X>\<P>:"
  value "Cons False Nil"
  value "rev(Cons True (Cons False Nil))"
  value "rev(Cons a (Cons b Nil))"

theorem rev_rev --"Reversing a list twice gives the original list."[simp]:
  "rev(rev xs) = xs"
--"\<P>\<F>:"
  apply(induction xs)
  apply(auto)
  oops                     (*Oops. Need a lemma.*)

lemma rev_app [simp]:
  "rev(app xs ys) = app(rev ys)(rev xs)"
--"\<P>\<F>:"
  apply(induction xs)
  apply(auto) oops         (*Oops. Auto doesn't even get rid of step 1.*)

lemma app_Nil2 [simp]:
  "app xs Nil = xs"
--"\<P>\<F>:"
  apply(induction xs)
  by(auto)                 (*Works. And it's added to simp.*)

lemma rev_app [simp]:
  "rev(app xs ys) = app(rev ys)(rev xs)"
--"\<P>\<F>:"
  apply(induction xs)
  apply(auto) oops         (*Oops. The lemma above in simp solves the base case.
                             But the inductive step has only been simplified to
                             a point where it needs associativity.*)

lemma app_assoc [simp]:
  "app (app xs ys) zs = app xs (app ys zs)"
--"\<P>\<F>:"
  apply(induction xs)
  by(auto)                 (*Works. Back to rev_app.*)

lemma rev_app [simp]:
  "rev(app xs ys) = app(rev ys)(rev xs)"
--"\<P>\<F>:"
  apply(induction xs)
  by(auto)                 (*Works. Back to rev_rev.*)

theorem rev_rev --"Reversing a list twice equals the original list."[simp]:
  "rev(rev xs) = xs"
--"\<P>\<F>:"
  apply(induction xs)
  by(auto)

--"Standard list operators."
  value "List.list.Nil"        (* "[]" :: "'a List.list" *)
  value "a::nat List.list"     (* "a" :: "nat List.list" *)
  value "(2::nat) # []"        (* "[2]"::"nat List.list". Same as "Cons 2 []".*)
  value "List.Cons (2::nat) []"       (* Same as above. *)
  value "List.Cons x xs"              (* "x # xs"::"'a List.list" *)
  value "[a,b,c] = a # b # c # []"    (* "True" :: "bool" *)
  value "(append xs ys) = (xs @ ys)"  (* "True" :: "bool" *)
  value "append xs ys"                (* "xs @ ys" :: "'a List.list" *)
  value "xs @ ys"
  value "[1::nat,2] @ [3,4]"
  value "append xs ys = xs @ ys"

fun add1 ::
  "nat => nat" where
--"\<F>\<U>:"
  "add1 x = x + 1"
--"\<X>\<P>:"
  value "[1::nat,2]"              (* "[1, 2]" :: "nat List.list" *)
  value "length"
  value "length [1::nat,2]"       (* "2" :: "nat" *)
```

```
  value "map"
  value "map add1 [1::nat,2]"     (* "[2, 3]" :: "nat List.list" *)

datatype ('a,'b)test2_3_1
  =Nil
  |Con "'a" "'b"

--"HOW TO PROVE INDUCTION."
(*For P, prove P(Nil). Then assume P(xs) and prove P(Cons x xs).*)
(*PG.13, To show P x for all x of type ('a1,...,'an)t:
  Assume: P(x\<^isub>j) for all j where t_i,j = ('a_1,...,'a_n)t.*)

datatype 'a tree --"As an example, consider binary trees [pg.13]."
  =Tip
  |Node "'a tree" 'a  "'a tree"
--"\<N>\<E>:"
    --"i12tu.{17}: size is 1 plus the sum of all the args of type t. The size of
       the args of Tip is 0, so (size Tip) is 1."
--"\<X>\<P>:"
  value "size Tip"
  value "size (Node Tip y Tip)"                          (*size = 1*)
  value "size (Node (Node Tip y Tip) y (Node Tip y Tip))" (*size = 3*)

fun mirror --"A mirror function for datatype tree, [pg.13]."::
  "'a tree \<Rightarrow> 'a tree" where
--"\<F>\<U>:"
  "mirror Tip = Tip" |
  "mirror (Node l a r) = Node (mirror r) a (mirror l)"

lemma --"The following lemma illustrates induction:"
  "mirror(mirror t) = t"
--"\<P>\<F>:"
  apply(induction t)
  by(auto)

definition sq --"Non recursive functions are defined as in this example."::
  "nat \<Rightarrow> nat" where
--"\<D>\<F>:"
  "sq n = n * n"
--"\<X>\<P>:"
  value "sq n"
  value "sq 20"

abbreviation sq' --"Abbreviations are similar to definitions:"::
  "nat \<Rightarrow> nat" where "sq' n == n * n"
--"\<X>\<P>:"
  value "(n::nat) * n"
  value "sq' (n::nat)"

fun div2 --"Functions defined with fun come with their own induction schema"::
  "nat \<Rightarrow> nat" where
--"\<F>\<U>:"
  "div2 0            = 0" |
  "div2 (Suc 0)      = Suc 0" |
  "div2 (Suc(Suc n)) = Suc(div2 n)"
--"\<X>\<P>:"
  (*The size of RHS arg of div2, n, is smaller than the RHS div2, Suc(Suc n).*)
  value "div2 1"
  value "div2 23"
  value "div2 ((n::nat)+n)"

lemma --"This customized induction rule can simplify inductive proofs."
  "div2(n+n) = n"
--"\<P>\<F>:"
  apply(induction n rule: div2.induct)
  by(auto)

lemma --"div2 using only apply(induction n)."
  "div2(n+n) = n"
--"\<P>\<F>:"
  apply(induction n)
  by(auto)
fun itrev --"A linear time version of rev"::
  "'a list \<Rightarrow> 'a list \<Rightarrow> 'a list" where
--"\<F>\<U>:"
  "itrev []     ys = ys" |
  "itrev (x#xs) ys = itrev xs (x#ys)" print_theorems
--"\<X>\<P>:"
  value "itrev [1::nat,2,3,4] []"
  value "itrev [] [1::nat,2,3,4]"
  value "itrev [1::nat,2,3,4] [8,9]"

lemma --"Only 1 variable, so the induction hypothesis is too weak."
  "itrev xs [] = rev xs"
  apply(induction xs)
  apply(auto)
  oops

lemma --"2 variables now, but only xs is suitable for induction."
  "itrev xs ys = rev xs @ ys"
  --"\<F>\<U>:"
    --"rev Nil = Nil"
    --"rev (Cons x xs) = app (rev xs) (Cons x Nil)"
  --"\<F>\<U>:"
    --"app Nil ys = ys"
    --"app (Cons x xs) ys = Cons x (app xs ys)"
  apply(induction xs, auto)
  oops
```

```
end
```