

(*ΝισαΘΞισαΠ*)

Notes

i12prI2.thy

Gottfried Barrow

August 6, 2012

Contents

Title	i
Table of Contents	ii
Theory Header	1
2 Programming and Proving [3.1_a]	1
2.1 Basics [3.2 _b]	1
2.1.1 Predefined Ops and Funcprog Constructs [3.4 _a]	1
...Sy.2.1.2....."Op * X Y" Is The Func For Predefined Binary Infix Functions [3.4 _a]	1
2.1.3 Lambda-Abstractions, Formulae, Equality, Quantifiers [4.1 _a]	1
...Ne.2.1.4....(abstractions).....Terms Can Contain Lambda Abstractions. [4.1 _a]	1
2.1.5 Not, And, Or, Imp, Quantifiers [4.2 _b]	1
...Op.2.1.6.....not, or, and, implication [4.1 _b]	1
...Op.2.1.7.....Equality "Op =" Means Iff For Formulae [4.1 _c]	1
...Op.2.1.8.....All, Exists [4.2 _d]	2
...Op.2.1.9.....Meta-logic: and, implication, equivalence [4.3 _e]	2
2.1.10 Right Arrows Associate Right and Imp Shorthand [4.3 _c]	2
...Sy.2.1.11.....meta-implication shorthand [4.3 _f]	2
2.2 Types bool, nat and list [5.3 _a]	2
2.2.1 Type bool [5.3 _b]	2
2.2.2 How conj could be defined [5.4 _c]	3
2.2.3 TYPE NAT [6.1 _a]	3
2.2.4 NAT ARITHMETIC AND COMPARISON FUNCTIONS [6.1 _b]	3
2.2.5 NAT ADD, A LEMMA AND PROOF [6.2 _c]	3
2.2.6 TYPE LIST [8.1 _a]	4
2.2.7 2.2.4 THE PROOF PROCESS [9.3 _a]	4
2.2.8 2.2.5 PREDEFINED LISTS [12.2 _a]	5
2.2.9 Standard list operators [12.2 _b]	5
2.3 Type and function definitions [12.3 _a]	6
2.3.1 General datatype definition [12.3 _b]	6
2.3.2 Structural induction for the general datatype [13.1 _a]	6
2.3.3 Definitions and Abbreviations [14.2 _a]	6
2.3.4 Recursive functions [14.3 _b]	7
2.4 Induction heuristics [16.1 _a]	7
Theory End	8
[—i12prI2.thy—]	9
[—i12prI2.thy \<cmds>—]	15

Theory Header

```

1 | theory i12prI2
2 | imports Complex_Main
3 |     "../..//pi/I"    (*Declare, print, sledge, nitP cmds. Not really needed.*)
4 | 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

```

5 | lemma testOp: "op ^ True True"
6 |   by auto

```

2.1.3 Lambda-Abstractions, Formulae, Equality, Quantifiers [4.1_a]

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

```

7 | lemma lambdaInTerm: "( $\lambda x. x$ ) a) = a"
8 |   by auto

```

2.1.5 Not, And, Or, Imp, Quantifiers [4.2_b]

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

```

9 | lemma notOp: " $\neg$ False"
10 |   by auto
11 | lemma andOp1: "op ^ True True"
12 |   by auto
13 | lemma andOp2: "True ^ True"
14 |   by auto
15 | lemma orOp: "op  $\vee$  True False"
16 |   by auto
17 | lemma impOp: "op  $\longrightarrow$  False True"
18 |   by auto
19 | lemma impOp2: "False  $\longrightarrow$  True"
20 |   by auto

```

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

```

21 | lemma equalityOp: "op = a a"
22 |   by auto
23 | lemma equalityIff: "(op = a a) = (op = b b)"
24 |   by auto

```

$(\omega \mid \text{OPERATOR})$ 2.1.8. All, Exists

```

25 lemma forallOp: "∀x. x=x"
26   by auto
27 lemma existsOp: "∃x. x=False"
28   by auto

```

 $(\omega \mid \text{OPERATOR})$ 2.1.9. Meta-logic: and, implication, equivalence

```

29 lemma metalogicAnd: "∧x. x = x"
30   by auto
31 lemma metalogicImp: "False ⇒ True"
32   by auto
33 lemma metalogicEqual: "a ≡ a"
34   by auto

```

2.1.10 Right Arrows Associate Right and Imp Shorthand [4.3.c]

 $(\sigma \mid \text{SYNTAX})$ 2.1.11. meta-implication shorthand

```

35 lemma impShorthand1: "[False;True] ⇒ True"
36   by auto
37 lemma impLonghand: "False ⇒ True ⇒ True"
38   by auto
39 lemma impShorthand4: "[/True;True/] ==> True"
40   by auto

```

2.2 Types bool, nat and list [5.3.a]

2.2.1 Type bool [5.3.b]

```

41 (*Keyword "datatype" defines an inductive datatype in HOL [pi12rRef.{213}]. It
42   requires Datatype.thy.*)
43 datatype boolTest = TrueTest | FalseTest
44 value "TrueTest" value "FalseTest"
45
46 (*However, though the tutorial shows that bool is defined by datatype, it's
47   actually defined with "typedecl bool" in HOL.thy.*)
48 typedecl boolTest2
49 (*judgment
50   Trueprop2      :: "boolTest2 => prop"*)
51 (*But I get this error unless commented out: "Attempt to redeclare object-logic
52   judgment". That's because there can only be one judgement per theory
53   [pi12rRef.{195}].*)
54 consts
55   TrueTest2      :: bool
56   FalseTest2    :: bool

```

2.2.2 How conj could be defined [5.4.c]

```

57 (*The keyword "fun" requires FunDef.thy.*)
58 fun conjTest :: "bool => bool => bool" where
59   "conjTest True True = True" |
60   "conjTest _ _      = False"
61 value "conjTest True True"
62 value "conj True True" (*using the normal conjunction*)

```

2.2.3 TYPE NAT [6.1.a]

```

63 value "0::nat"
64 value "Suc(0::nat)"

```

2.2.4 NAT ARITHMETIC AND COMPARISON FUNCTIONS [6.1.b]

```

65 value "0 ≤ Suc 0"
66 value "0 ≥ Suc 0"
67 value "Suc(Suc(Suc 0))"
68 value "Suc(Suc(Suc 0)) + Suc(Suc(Suc 0))"
69 value "Suc(Suc(Suc 0)) - Suc(Suc(Suc 0))"
70 value "Suc(Suc 0) - Suc(Suc(Suc 0))"
71 value "Suc 0 = 1"
72 value "Suc(Suc(Suc 0)) + Suc(Suc(Suc 0)) = 6"
73 value "Suc(Suc(Suc 0)) * Suc(Suc(Suc 0)) * Suc(Suc(Suc 0))
74       = 3^3"

```

2.2.5 NAT ADD, A LEMMA AND PROOF [6.2.c]

```

75 fun add :: "nat => nat => nat" where
76   "add 0 n = n" |
77   "add (Suc m) n = Suc(add m n)"
78 value "add 1 2::nat"
79 value "add m 0 = m"
80
81 theorem add: "add m 0 = m"
82   (*JV PROOF:
83     BASIS: Let m=0. Then 0+0=0, by the first condition of addT definition.
84     INDUCTION STEP: Assume that m+0=m. Need to show that (Suc m) + 0 = Suc m.
85     By definition of addT and m+0=m, (Suc m)+0 = Suc(m+0) = Suc m.
86   *)
87   apply(induction m)
88   apply(auto)
89   done
90
91 thm add
92
93 lemma stuffer1: "x + 0 = x"
94   oops
95 lemma stuffer2: "x + (0::nat) = x"
96   oops
97
98 value "0";

```

2.2.6 TYPE LIST [8.1.a]

```

99
100 datatype 'a list2 --"List type. Renamed to not clash with List.list."
101   =Nil2
102   |Cons2 'a "'a list2"
103 --"χφ:"
104   value "Cons (2::nat) (Nil::nat list)"
105   value "Cons (2::nat) (Nil)"
106   value "Cons d (Cons c (Cons a (Cons b (Nil))))"
107   value "Cons x Nil"
108   value "Nil = Nil"
109   value "Cons x y"
110
111 fun app --"Append one list to another list [pg.8]"::
112   "'a list ⇒ 'a list ⇒ 'a list" where
113 --"ℱℳ:"
114   "app Nil ys = ys" |
115   "app (Cons x xs) ys = Cons x (app xs ys)"
116 --"χφ:"
117   value "app (Cons x Nil) Nil"
118   value "app (Cons x Nil) (Cons x Nil)"
119
120 fun rev --"Reverse a list."::
121   "'a list ⇒ 'a list" where
122 --"ℱℳ:"
123   "rev Nil = Nil" |
124   "rev (Cons x xs) = app (rev xs) (Cons x Nil)"
125 --"χφ:"
126   value "Cons False Nil"
127   value "rev(Cons True (Cons False Nil))"
128   value "rev(Cons a (Cons b Nil))"

```

2.2.7 2.2.4 THE PROOF PROCESS [9.3.a]

```

129
130 theorem rev_rev --"Reversing a list twice gives the original list."[simp]:
131   "rev(rev xs) = xs"
132 --"φℱ:"
133   apply(induction xs)
134   apply(auto)
135   oops (*Oops. Need a lemma.*)
136
137 lemma rev_app [simp]:
138   "rev(app xs ys) = app(rev ys)(rev xs)"
139 --"φℱ:"
140   apply(induction xs)
141   apply(auto) oops (*Oops. Auto doesn't even get rid of step 1.*)
142
143 lemma app_Nil2 [simp]:
144   "app xs Nil = xs"
145 --"φℱ:"
146   apply(induction xs)
147   by(auto) (*Works. And it's added to simp.*)
148
149 lemma rev_app [simp]:
150   "rev(app xs ys) = app(rev ys)(rev xs)"

```

```

151 --"PF:"
152   apply(induction xs)
153   apply(auto) oops          (*Oops. The lemma above in simp solves the base case.
154                             But the inductive step has only been simplified to
155                             a point where it needs associativity.*)
156
157 lemma app_assoc [simp]:
158   "app (app xs ys) zs = app xs (app ys zs)"
159 --"PF:"
160   apply(induction xs)
161   by(auto)                  (*Works. Back to rev_app.*)
162
163 lemma rev_app [simp]:
164   "rev(app xs ys) = app(rev ys)(rev xs)"
165 --"PF:"
166   apply(induction xs)
167   by(auto)                  (*Works. Back to rev_rev.*)
168
169 theorem rev_rev --"Reversing a list twice equals the original list."[simp]:
170   "rev(rev xs) = xs"
171 --"PF:"
172   apply(induction xs)
173   by(auto)

```

2.2.8 2.2.5 PREDEFINED LISTS [12.2.a]

2.2.9 Standard list operators [12.2.b]

```

174
175 --"Standard list operators."
176 value "List.list.Nil"          (* "[]" :: "'a List.list" *)
177 value "a::nat List.list"      (* "a" :: "nat List.list" *)
178 value "(2::nat) # []"         (* "[2]" :: "nat List.list". Same as "Cons 2 []".*)
179 value "List.Cons (2::nat) []" (* Same as above. *)
180 value "List.Cons x xs"        (* "x # xs" :: "'a List.list" *)
181 value "[a,b,c] = a # b # c # []" (* "True" :: "bool" *)
182 value "(append xs ys) = (xs @ ys)" (* "True" :: "bool" *)
183 value "append xs ys"         (* "xs @ ys" :: "'a List.list" *)
184 value "xs @ ys"
185 value "[1::nat,2] @ [3,4]"
186 value "append xs ys = xs @ ys"
187
188 fun add1 ::
189   "nat => nat" where
190 --"FU:"
191   "add1 x = x + 1"
192 --"XP:"
193 value "[1::nat,2]"            (* "[1, 2]" :: "nat List.list" *)
194 value "length"
195 value "length [1::nat,2]"     (* "2" :: "nat" *)
196 value "map"
197 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]

```

198
199 datatype ('a,'b)test2_3_1
200   =Nil
201   |Con "'a" "'b"

```

2.3.2 Structural induction for the general datatype [13.1.a]

```

202
203 --"HOW TO PROVE INDUCTION."
204 (*For P, prove P(Nil). Then assume P(xs) and prove P(Cons x xs).*)
205 (*PG.13, To show P x for all x of type ('a1,...,'an)t:
206   Assume: P(xj) for all j where ti,j = ('a1,..., 'an)t.*)
207
208 datatype 'a tree --"As an example, consider binary trees [pg.13]."
209   =Tip
210   |Node "'a tree" 'a "'a tree"
211 --"NE:"
212   --"i12tu.{17}: size is 1 plus the sum of all the args of type t. The size of
213     the args of Tip is 0, so (size Tip) is 1."
214 --"XP:"
215   value "size Tip"
216   value "size (Node Tip y Tip)" (*size = 1*)
217   value "size (Node (Node Tip y Tip) y (Node Tip y Tip))" (*size = 3*)
218
219 fun mirror --"A mirror function for datatype tree, [pg.13]."::
220   "'a tree ⇒ 'a tree" where
221 --"FU:"
222   "mirror Tip = Tip" |
223   "mirror (Node l a r) = Node (mirror r) a (mirror l)"
224
225 lemma --"The following lemma illustrates induction:"
226   "mirror(mirror t) = t"
227 --"PF:"
228   apply(induction t)
229   by(auto)

```

2.3.3 Definitions and Abbreviations [14.2.a]

```

230
231 definition sq --"Non recursive functions are defined as in this example."::
232   "nat ⇒ nat" where
233 --"DF:"
234   "sq n = n * n"
235 --"XP:"
236   value "sq n"
237   value "sq 20"
238
239 abbreviation sq' --"Abbreviations are similar to definitions:"::
240   "nat ⇒ nat" where "sq' n == n * n"
241 --"XP:"
242   value "(n::nat) * n"
243   value "sq' (n::nat)"

```


2.3.4 Recursive functions [14.3.b]

```

244
245 fun div2 --"Functions defined with fun come with their own induction schema"::
246   "nat ⇒ nat" where
247 --"FU:"
248   "div2 0          = 0" |
249   "div2 (Suc 0)    = Suc 0" |
250   "div2 (Suc(Suc n)) = Suc(div2 n)"
251 --"XP:"
252   (*The size of RHS arg of div2, n, is smaller than the RHS div2, Suc(Suc n).*)
253   value "div2 1"
254   value "div2 23"
255   value "div2 ((n::nat)+n)"
256
257 lemma --"This customized induction rule can simplify inductive proofs."
258   "div2(n+n) = n"
259 --"PF:"
260   apply(induction n rule: div2.induct)
261   by(auto)
262
263 lemma --"div2 using only apply(induction n)."
264   "div2(n+n) = n"
265 --"PF:"
266   apply(induction n)
267   by(auto)

```

2.4 Induction heuristics [16.1.a]

```

268 fun itrev --"A linear time version of rev"::
269   "'a list ⇒ 'a list ⇒ 'a list" where
270 --"FU:"
271   "itrev []      ys = ys" |
272   "itrev (x#xs) ys = itrev xs (x#ys)" print_theorems
273 --"XP:"
274   value "itrev [1::nat,2,3,4] []"
275   value "itrev [] [1::nat,2,3,4]"
276   value "itrev [1::nat,2,3,4] [8,9]"
277
278 lemma --"Only 1 variable, so the induction hypothesis is too weak."
279   "itrev xs [] = rev xs"
280   apply(induction xs)
281   apply(auto)
282   oops
283
284 lemma --"2 variables now, but only xs is suitable for induction."
285   "itrev xs ys = rev xs @ ys"
286 --"FU:"
287   --"rev Nil = Nil"
288   --"rev (Cons x xs) = app (rev xs) (Cons x Nil)"
289 --"FU:"
290   --"app Nil ys = ys"
291   --"app (Cons x xs) ys = Cons x (app xs ys)"
292   apply(induction xs, auto)
293   oops
294
295

```

296 |

Theory End

297 | *end*

[—i12prI2.thy—]

```

1 theory i12prI2
2 imports Complex_Main
3     ".././pi/I" (*Declare, print, sledge, nitP cmds. Not really needed.*)
4 begin
5 lemma testOp: "op  $\wedge$  True True"
6   by auto
7 lemma lambdaInTerm: " $((\lambda x. x) a) = a$ "
8   by auto
9 lemma notOp: " $\neg$ False"
10  by auto
11 lemma andOp1: "op  $\wedge$  True True"
12  by auto
13 lemma andOp2: "True  $\wedge$  True"
14  by auto
15 lemma orOp: "op  $\vee$  True False"
16  by auto
17 lemma impOp: "op  $\longrightarrow$  False True"
18  by auto
19 lemma impOp2: "False  $\longrightarrow$  True"
20  by auto
21 lemma equalityOp: "op = a a"
22  by auto
23 lemma equalityIff: "(op = a a) = (op = b b)"
24  by auto
25 lemma forallOp: " $\forall x. x=x$ "
26  by auto
27 lemma existsOp: " $\exists x. x=False$ "
28  by auto
29 lemma metalogicAnd: " $\wedge x. x = x$ "
30  by auto
31 lemma metalogicImp: "False  $\implies$  True"
32  by auto
33 lemma metalogicEqual: "a  $\equiv$  a"
34  by auto
35 lemma impShorthand1: "[False;True]  $\implies$  True"
36  by auto
37 lemma impLonghand: "False  $\implies$  True  $\implies$  True"
38  by auto
39 lemma impShorthand4: "[True;True]  $\implies$  True"
40  by auto
41 (*Keyword "datatype" defines an inductive datatype in HOL [pi12rRef.{213}]. It
42   requires Datatype.thy.*)
43 datatype boolTest = TrueTest | FalseTest
44 value "TrueTest" value "FalseTest"
45
46 (*However, though the tutorial shows that bool is defined by datatype, it's
47   actually defined with "typeddecl bool" in HOL.thy.*)
48 typeddecl boolTest2
49 (*judgment
50   Trueprop2      :: "boolTest2 => prop"*)
51   (*But I get this error unless commented out: "Attempt to redeclare object-logic
52     judgment". That's because there can only be one judgement per theory
53     [pi12rRef.{195}].*)
54 consts
55   TrueTest2      :: bool
56   FalseTest2     :: bool

```

```

57 (*The keyword "fun" requires FunDef.thy.*)
58 fun conjTest :: "bool => bool => bool" where
59   "conjTest True True = True" |
60   "conjTest _ _      = False"
61 value "conjTest True True"
62 value "conj True True" (*using the normal conjunction*)
63 value "0::nat"
64 value "Suc(0::nat)"
65 value "0 ≤ Suc 0"
66 value "0 ≥ Suc 0"
67 value "Suc(Suc(Suc 0))"
68 value "Suc(Suc(Suc 0)) + Suc(Suc(Suc 0))"
69 value "Suc(Suc(Suc 0)) - Suc(Suc(Suc 0))"
70 value "Suc(Suc 0) - Suc(Suc(Suc 0))"
71 value "Suc 0 = 1"
72 value "Suc(Suc(Suc 0)) + Suc(Suc(Suc 0)) = 6"
73 value "Suc(Suc(Suc 0)) * Suc(Suc(Suc 0)) * Suc(Suc(Suc 0))
74       = 3^3"
75 fun add :: "nat => nat => nat" where
76   "add 0 n = n" |
77   "add (Suc m) n = Suc(add m n)"
78 value "add 1 2::nat"
79 value "add m 0 = m"
80
81 theorem add: "add m 0 = m"
82   (*JV PROOF:
83     BASIS: Let m=0. Then 0+0=0, by the first condition of addT definition.
84     INDUCTION STEP: Assume that m+0=m. Need to show that (Suc m) + 0 = Suc m.
85     By definition of addT and m+0=m, (Suc m)+0 = Suc(m+0) = Suc m.
86   *)
87   apply(induction m)
88   apply(auto)
89   done
90
91 thm add
92
93 lemma stuffer1: "x + 0 = x"
94   oops
95 lemma stuffer2: "x + (0::nat) = x"
96   oops
97
98 value "0";
99
100 datatype 'a list2 --"List type. Renamed to not clash with List.list."
101   =Nil2
102   |Cons2 'a "'a list2"
103 --"λφ:"
104   value "Cons (2::nat) (Nil::nat list)"
105   value "Cons (2::nat) (Nil)"
106   value "Cons d (Cons c (Cons a (Cons b (Nil))))"
107   value "Cons x Nil"
108   value "Nil = Nil"
109   value "Cons x y"
110
111 fun app --"Append one list to another list [pg.8]"::
112   "'a list ⇒ 'a list ⇒ 'a list" where
113 --"ℱU:"
114   "app Nil ys = ys" |

```

```

115 "app (Cons x xs) ys = Cons x (app xs ys)"
116 --"X $\mathcal{P}$ :"
117 value "app (Cons x Nil) Nil"
118 value "app (Cons x Nil) (Cons x Nil)"
119
120 fun rev --"Reverse a list."::
121 "'a list  $\Rightarrow$  'a list" where
122 --"F $\mathcal{U}$ :"
123 "rev Nil = Nil" |
124 "rev (Cons x xs) = app (rev xs) (Cons x Nil)"
125 --"X $\mathcal{P}$ :"
126 value "Cons False Nil"
127 value "rev(Cons True (Cons False Nil))"
128 value "rev(Cons a (Cons b Nil))"
129
130 theorem rev_rev --"Reversing a list twice gives the original list."[simp]:
131 "rev(rev xs) = xs"
132 --"P $\mathcal{F}$ :"
133 apply(induction xs)
134 apply(auto)
135 oops (*Oops. Need a lemma.*)
136
137 lemma rev_app [simp]:
138 "rev(app xs ys) = app(rev ys)(rev xs)"
139 --"P $\mathcal{F}$ :"
140 apply(induction xs)
141 apply(auto) oops (*Oops. Auto doesn't even get rid of step 1.*)
142
143 lemma app_Nil2 [simp]:
144 "app xs Nil = xs"
145 --"P $\mathcal{F}$ :"
146 apply(induction xs)
147 by(auto) (*Works. And it's added to simp.*)
148
149 lemma rev_app [simp]:
150 "rev(app xs ys) = app(rev ys)(rev xs)"
151 --"P $\mathcal{F}$ :"
152 apply(induction xs)
153 apply(auto) oops (*Oops. The lemma above in simp solves the base case.
154 But the inductive step has only been simplified to
155 a point where it needs associativity.*)
156
157 lemma app_assoc [simp]:
158 "app (app xs ys) zs = app xs (app ys zs)"
159 --"P $\mathcal{F}$ :"
160 apply(induction xs)
161 by(auto) (*Works. Back to rev_app.*)
162
163 lemma rev_app [simp]:
164 "rev(app xs ys) = app(rev ys)(rev xs)"
165 --"P $\mathcal{F}$ :"
166 apply(induction xs)
167 by(auto) (*Works. Back to rev_rev.*)
168
169 theorem rev_rev --"Reversing a list twice equals the original list."[simp]:
170 "rev(rev xs) = xs"
171 --"P $\mathcal{F}$ :"
172 apply(induction xs)

```

```

173 by(auto)
174
175 --"Standard list operators."
176 value "List.list.Nil" (* "[]" :: "'a List.list" *)
177 value "a::nat List.list" (* "a" :: "nat List.list" *)
178 value "(2::nat) # []" (* "[2]::"nat List.list". Same as "Cons 2 []".*)
179 value "List.Cons (2::nat) []" (* Same as above. *)
180 value "List.Cons x xs" (* "x # xs"::"'a List.list" *)
181 value "[a,b,c] = a # b # c # []" (* "True" :: "bool" *)
182 value "(append xs ys) = (xs @ ys)" (* "True" :: "bool" *)
183 value "append xs ys" (* "xs @ ys" :: "'a List.list" *)
184 value "xs @ ys"
185 value "[1::nat,2] @ [3,4]"
186 value "append xs ys = xs @ ys"
187
188 fun add1 ::
189 "nat => nat" where
190 --"FU:"
191 "add1 x = x + 1"
192 --"XP:"
193 value "[1::nat,2]" (* "[1, 2]" :: "nat List.list" *)
194 value "length"
195 value "length [1::nat,2]" (* "2" :: "nat" *)
196 value "map"
197 value "map add1 [1::nat,2]" (* "[2, 3]" :: "nat List.list" *)
198
199 datatype ('a,'b)test2_3_1
200 =Nil
201 |Con "'a" "'b"
202
203 --"HOW TO PROVE INDUCTION."
204 (*For P, prove P(Nil). Then assume P(xs) and prove P(Cons x xs).*)
205 (*PG.13, To show P x for all x of type ('a1,...,'an)t:
206 Assume: P(xj) for all j where ti,j = ('a1,..., 'an)t.
207
208 datatype 'a tree --"As an example, consider binary trees [pg.13].
209 =Tip
210 |Node "'a tree" 'a "'a tree"
211 --"NE:"
212 --"i12tu.{17}: size is 1 plus the sum of all the args of type t. The size of
213 the args of Tip is 0, so (size Tip) is 1."
214 --"XP:"
215 value "size Tip"
216 value "size (Node Tip y Tip)" (*size = 1*)
217 value "size (Node (Node Tip y Tip) y (Node Tip y Tip))" (*size = 3*)
218
219 fun mirror --"A mirror function for datatype tree, [pg.13].":
220 "'a tree => 'a tree" where
221 --"FU:"
222 "mirror Tip = Tip" |
223 "mirror (Node l a r) = Node (mirror r) a (mirror l)"
224
225 lemma --"The following lemma illustrates induction:"
226 "mirror(mirror t) = t"
227 --"PF:"
228 apply(induction t)
229 by(auto)
230

```

```

231 definition sq --"Non recursive functions are defined as in this example."::
232   "nat ⇒ nat" where
233   --"DF:"
234   "sq n = n * n"
235   --"XP:"
236   value "sq n"
237   value "sq 20"
238
239 abbreviation sq' --"Abbreviations are similar to definitions:"::
240   "nat ⇒ nat" where "sq' n == n * n"
241   --"XP:"
242   value "(n::nat) * n"
243   value "sq' (n::nat)"
244
245 fun div2 --"Functions defined with fun come with their own induction schema":
246   "nat ⇒ nat" where
247   --"FU:"
248   "div2 0          = 0" |
249   "div2 (Suc 0)    = Suc 0" |
250   "div2 (Suc(Suc n)) = Suc(div2 n)"
251   --"XP:"
252   (*The size of RHS arg of div2, n, is smaller than the RHS div2, Suc(Suc n).*)
253   value "div2 1"
254   value "div2 23"
255   value "div2 ((n::nat)+n)"
256
257 lemma --"This customized induction rule can simplify inductive proofs."
258   "div2(n+n) = n"
259   --"PF:"
260   apply(induction n rule: div2.induct)
261   by(auto)
262
263 lemma --"div2 using only apply(induction n)."
264   "div2(n+n) = n"
265   --"PF:"
266   apply(induction n)
267   by(auto)
268 fun itrev --"A linear time version of rev":
269   "'a list ⇒ 'a list ⇒ 'a list" where
270   --"FU:"
271   "itrev []      ys = ys" |
272   "itrev (x#xs) ys = itrev xs (x#ys)" print_theorems
273   --"XP:"
274   value "itrev [1::nat,2,3,4] []"
275   value "itrev [] [1::nat,2,3,4]"
276   value "itrev [1::nat,2,3,4] [8,9]"
277
278 lemma --"Only 1 variable, so the induction hypothesis is too weak."
279   "itrev xs [] = rev xs"
280   apply(induction xs)
281   apply(auto)
282   oops
283
284 lemma --"2 variables now, but only xs is suitable for induction."
285   "itrev xs ys = rev xs @ ys"
286   --"FU:"
287   --"rev Nil = Nil"
288   --"rev (Cons x xs) = app (rev xs) (Cons x Nil)"

```

```
289 | --" $\mathcal{F}U$ :"
290 |   --"app Nil ys = ys"
291 |   --"app (Cons x xs) ys = Cons x (app xs ys)"
292 | apply(induction xs, auto)
293 | oops
294 |
295 |
296 |
297 | end
```


[—i12prI2.thy \<cmds>—]

```

1 | theory i12prI2
2 | imports Complex_Main
3 |   ".../pi/I" (*Declare, print, sledge, nitP cmds. Not really needed.*)
4 | begin
5 | lemma testOp: "op \<and> True True"
6 |   by auto
7 | lemma lambdaInTerm: "((\<lambda>x. x) a) = a"
8 |   by auto
9 | lemma notOp: "\<not>False"
10 |   by auto
11 | lemma andOp\<isub>\<alpha>: "op \<and> True True"
12 |   by auto
13 | lemma andOp\<isub>2: "True \<and> True"
14 |   by auto
15 | lemma orOp: "op \<or> True False"
16 |   by auto
17 | lemma impOp: "op \<longrightrightarrow> False True"
18 |   by auto
19 | lemma impOp\<isub>2: "False \<longrightrightarrow> True"
20 |   by auto
21 | lemma equalityOp: "op = a a"
22 |   by auto
23 | lemma equalityIff: "(op = a a) = (op = b b)"
24 |   by auto
25 | lemma forallOp: "\<forall>x. x=x"
26 |   by auto
27 | lemma existsOp: "\<exists>x. x=False"
28 |   by auto
29 | lemma metalogicAnd: "\<And>x. x = x"
30 |   by auto
31 | lemma metalogicImp: "False \<Longrightrightarrow> True"
32 |   by auto
33 | lemma metalogicEqual: "a \<equiv> a"
34 |   by auto
35 | lemma impShorthand1: "\<lbrakk>False;True\<rbrakk> \<Longrightrightarrow> True"
36 |   by auto
37 | lemma impLonghand: "False \<Longrightrightarrow> True \<Longrightrightarrow> True"
38 |   by auto
39 | lemma impShorthand4: "[|True;True|] ==> True"
40 |   by auto
41 | (*Keyword "datatype" defines an inductive datatype in HOL [pi12rRef.{213}]. It
42 |   requires Datatype.thy.*)
43 | datatype boolTest = TrueTest | FalseTest
44 | value "TrueTest" value "FalseTest"
45 |
46 | (*However, though the tutorial shows that bool is defined by datatype, it's
47 |   actually defined with "typedec1 bool" in HOL.thy.*)
48 | typedec1 boolTest2
49 | (*judgment
50 |   Trueprop2      :: "boolTest2 => prop"*)
51 | (*But I get this error unless commented out: "Attempt to redeclare object-logic
52 |   judgment". That's because there can only be one judgement per theory
53 |   [pi12rRef.{195}].*)
54 | consts
55 |   TrueTest2      :: bool
56 |   FalseTest2     :: bool
57 | (*The keyword "fun" requires FunDef.thy.*)
58 | fun conjTest :: "bool => bool => bool" where
59 |   "conjTest True True = True" |
60 |   "conjTest _ _ = False"
61 | value "conjTest True True" (*using the normal conjunction*)
62 | value "conj True True"
63 | value "0::nat"
64 | value "Suc(0::nat)"
65 | value "0 \<le> Suc 0"
66 | value "0 \<ge> Suc 0"
67 | value "Suc(Suc(Suc 0))"
68 | value "Suc(Suc(Suc 0)) + Suc(Suc(Suc 0))"
69 | value "Suc(Suc(Suc 0)) - Suc(Suc(Suc 0))"
70 | value "Suc(Suc 0) - Suc(Suc(Suc 0))"
71 | value "Suc 0 = 1"
72 | value "Suc(Suc(Suc 0)) + Suc(Suc(Suc 0)) = 6"
73 | value "Suc(Suc(Suc 0)) * Suc(Suc(Suc 0)) * Suc(Suc(Suc 0))
74 |   = 3^3"
75 | fun add :: "nat => nat => nat" where
76 |   "add 0 n = n" |
77 |   "add (Suc m) n = Suc(add m n)"
78 | value "add 1 2::nat"
79 | value "add m 0 = m"
80 |
81 | theorem add: "add m 0 = m"
82 |   (*JV PROOF:
83 |     BASIS: Let m=0. Then 0+0=0, by the first condition of addT definition.
84 |     INDUCTION STEP: Assume that m+0=m. Need to show that (Suc m) + 0 = Suc m.
85 |     By definition of addT and m+0=m, (Suc m)+0 = Suc(m+0) = Suc m.
86 |   *)
87 |   apply(induction m)
88 |   apply(auto)
89 |   done
90 |
91 | thm add
92 |
93 | lemma stuffer1: "x + 0 = x"
94 |   oops
95 | lemma stuffer2: "x + (0::nat) = x"
96 |   oops

```

```

97
98 value "0";
99
100 datatype 'a list2 --"List type. Renamed to not clash with List.list."
101   =Nil2
102   |Cons2 'a "'a list2"
103   --"\<X>\<P>:"
104   value "Cons (2::nat) (Nil::nat list)"
105   value "Cons (2::nat) (Nil)"
106   value "Cons d (Cons c (Cons a (Cons b (Nil))))"
107   value "Cons x Nil"
108   value "Nil = Nil"
109   value "Cons x y"
110
111 fun app --"Append one list to another list [pg.8]"::
112   "a list \<Rightarrow> 'a list \<Rightarrow> 'a list" where
113   --"\<F>\<U>:"
114   "app Nil ys = ys" |
115   "app (Cons x xs) ys = Cons x (app xs ys)"
116   --"\<X>\<P>:"
117   value "app (Cons x Nil) Nil"
118   value "app (Cons x Nil) (Cons x Nil)"
119
120 fun rev --"Reverse a list."::
121   "a list \<Rightarrow> 'a list" where
122   --"\<F>\<U>:"
123   "rev Nil = Nil" |
124   "rev (Cons x xs) = app (rev xs) (Cons x Nil)"
125   --"\<X>\<P>:"
126   value "Cons False Nil"
127   value "rev(Cons True (Cons False Nil))"
128   value "rev(Cons a (Cons b Nil))"
129
130 theorem rev_rev --"Reversing a list twice gives the original list."[simp]:
131   "rev(rev xs) = xs"
132   --"\<P>\<F>:"
133   apply(induction xs)
134   apply(auto)
135   oops (*Oops. Need a lemma.*)
136
137 lemma rev_app [simp]:
138   "rev(app xs ys) = app(rev ys)(rev xs)"
139   --"\<P>\<F>:"
140   apply(induction xs)
141   apply(auto) oops (*Oops. Auto doesn't even get rid of step 1.*)
142
143 lemma app_Nil2 [simp]:
144   "app xs Nil = xs"
145   --"\<P>\<F>:"
146   apply(induction xs)
147   by(auto) (*Works. And it's added to simp.*)
148
149 lemma rev_app [simp]:
150   "rev(app xs ys) = app(rev ys)(rev xs)"
151   --"\<P>\<F>:"
152   apply(induction xs)
153   apply(auto) oops (*Oops. The lemma above in simp solves the base case.
154   But the inductive step has only been simplified to
155   a point where it needs associativity.*)
156
157 lemma app_assoc [simp]:
158   "app (app xs ys) zs = app xs (app ys zs)"
159   --"\<P>\<F>:"
160   apply(induction xs)
161   by(auto) (*Works. Back to rev_app.*)
162
163 lemma rev_app [simp]:
164   "rev(app xs ys) = app(rev ys)(rev xs)"
165   --"\<P>\<F>:"
166   apply(induction xs)
167   by(auto) (*Works. Back to rev_rev.*)
168
169 theorem rev_rev --"Reversing a list twice equals the original list."[simp]:
170   "rev(rev xs) = xs"
171   --"\<P>\<F>:"
172   apply(induction xs)
173   by(auto)
174
175 --"Standard list operators."
176 value "List.list.Nil" (* "[ ] :: 'a List.list" *)
177 value "a::nat List.list" (* "a" :: "nat List.list" *)
178 value "(2::nat) # [ ]" (* "[2]::"nat List.list". Same as "Cons 2 [ ]".*)
179 value "List.Cons (2::nat) [ ]" (* Same as above. *)
180 value "List.Cons x xs" (* "x # xs"::"'a List.list" *)
181 value "[a,b,c] = a # b # c # [ ]" (* "True" :: "bool" *)
182 value "(append xs ys) = (xs @ ys)" (* "True" :: "bool" *)
183 value "append xs ys" (* "xs @ ys" :: "'a List.list" *)
184 value "xs @ ys"
185 value "[1::nat,2] @ [3,4]"
186 value "append xs ys = xs @ ys"
187
188 fun add1 ::
189   "nat => nat" where
190   --"\<F>\<U>:"
191   "add1 x = x + 1"
192   --"\<X>\<P>:"
193   value "[1::nat,2]" (* "[1, 2] :: "nat List.list" *)
194   value "length"
195   value "length [1::nat,2]" (* "2" :: "nat" *)

```

```

196 value "map"
197 value "map add1 [1::nat,2]" (* "[2, 3]" :: "nat List.list" *)
198
199 datatype ('a,'b)test2_3_1
200 =Nil
201 |Con "'a" "'b"
202
203 --"HOW TO PROVE INDUCTION."
204 (*For P, prove P(Nil). Then assume P(xs) and prove P(Cons x xs).*)
205 (*PG.13, To show P x for all x of type ('a1,...,'an)t:
206 Assume: P(x\<isub>j) for all j where t_i,j = ('a_1,...,'a_n)t.*
207
208 datatype 'a tree --"As an example, consider binary trees [pg.13]."
209 =Tip
210 |Node "'a tree" 'a "'a tree"
211 --"\<N>\<E>:"
212 --"i12tu.{17}: size is 1 plus the sum of all the args of type t. The size of
213 the args of Tip is 0, so (size Tip) is 1."
214 --"\<X>\<P>:"
215 value "size Tip"
216 value "size (Node Tip y Tip)" (*size = 1*)
217 value "size (Node (Node Tip y Tip) y (Node Tip y Tip))" (*size = 3*)
218
219 fun mirror --"A mirror function for datatype tree, [pg.13].":
220 "a tree \<Rightarrow> 'a tree" where
221 --"\<F>\<U>:"
222 "mirror Tip = Tip" |
223 "mirror (Node l a r) = Node (mirror r) a (mirror l)"
224
225 lemma --"The following lemma illustrates induction:"
226 "mirror(mirror t) = t"
227 --"\<P>\<F>:"
228 apply(induction t)
229 by(auto)
230
231 definition sq --"Non recursive functions are defined as in this example.":
232 "nat \<Rightarrow> nat" where
233 --"\<D>\<F>:"
234 "sq n = n * n"
235 --"\<X>\<P>:"
236 value "sq n"
237 value "sq 20"
238
239 abbreviation sq' --"Abbreviations are similar to definitions.":
240 "nat \<Rightarrow> nat" where "sq' n == n * n"
241 --"\<X>\<P>:"
242 value "(n::nat) * n"
243 value "sq' (n::nat)"
244
245 fun div2 --"Functions defined with fun come with their own induction schema":
246 "nat \<Rightarrow> nat" where
247 --"\<F>\<U>:"
248 "div2 0 = 0" |
249 "div2 (Suc 0) = Suc 0" |
250 "div2 (Suc(Suc n)) = Suc(div2 n)"
251 --"\<X>\<P>:"
252 (*The size of RHS arg of div2, n, is smaller than the RHS div2, Suc(Suc n).*)
253 value "div2 1"
254 value "div2 23"
255 value "div2 ((n::nat)+n)"
256
257 lemma --"This customized induction rule can simplify inductive proofs."
258 "div2(n+n) = n"
259 --"\<P>\<F>:"
260 apply(induction n rule: div2.induct)
261 by(auto)
262
263 lemma --"div2 using only apply(induction n)."
264 "div2(n+n) = n"
265 --"\<P>\<F>:"
266 apply(induction n)
267 by(auto)
268
269 fun itrev --"A linear time version of rev":
270 "'a list \<Rightarrow> 'a list \<Rightarrow> 'a list" where
271 --"\<F>\<U>:"
272 "itrev [] ys = ys" |
273 "itrev (x#xs) ys = itrev xs (x#ys)" print_theorems
274 --"\<X>\<P>:"
275 value "itrev [1::nat,2,3,4] []"
276 value "itrev [] [1::nat,2,3,4]"
277 value "itrev [1::nat,2,3,4] [8,9]"
278
279 lemma --"Only 1 variable, so the induction hypothesis is too weak."
280 "itrev xs [] = rev xs"
281 apply(induction xs)
282 apply(auto)
283 oops
284
285 lemma --"2 variables now, but only xs is suitable for induction."
286 "itrev xs ys = rev xs @ ys"
287 --"\<F>\<U>:"
288 --"rev Nil = Nil"
289 --"rev (Cons x xs) = app (rev xs) (Cons x Nil)"
290 --"\<F>\<U>:"
291 --"app Nil ys = ys"
292 --"app (Cons x xs) ys = Cons x (app xs ys)"
293 apply(induction xs, auto)
294 oops

```

295 |
296 |
297 | *end*