ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Dipl.-Inf. Achim D. Brucker
Dr. Burkhart Wolff

Computer-supported Modeling and Reasoning

http://www.infsec.ethz.ch/education/permanent/csmr/
(rev. 16814)
Submission date: –

HOL: Specifying and Proving AVL-Trees

This exercises describes a small modeling and verification project going over two weeks. We will specify and analyze a widely-used data structure AVL-trees as a (purely functional) implementation.

As proof techniques, we will use automatic case splitting in the simplifier supporting reasoning over recursive functions with pattern matching (**recdef**). Finally, we use Isabelle's code generator the convert the function definitions into "real" SML programs.

1. The Problem: AVL trees

In 1962 Adel'son-Vel'skiĭ and Landis introduced a class of balanced binary search trees (called AVL trees) that guarantee that a tree with n internal nodes has height $O(\log n)$. The efficiency of AVL tree hinges on the fact that a tree should be *balanced* and *ordered*. Of course, when a node is inserted into or deleted from the tree, these properties must be maintained, by certain rotation operations on AVL trees. Note that unless a tree contains $2^n - 1$ nodes for some n, it cannot be "exactly" balanced. All we can expect is that the height of the left and right subtrees differ by at most one. In order to decide in which way a tree should be rotated, it is convenient to have a function *bal* that tells us if a

AVL

bal

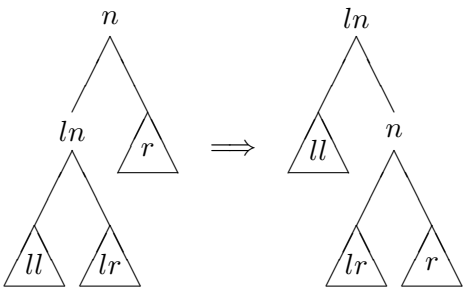

Figure 1: *r_rot*

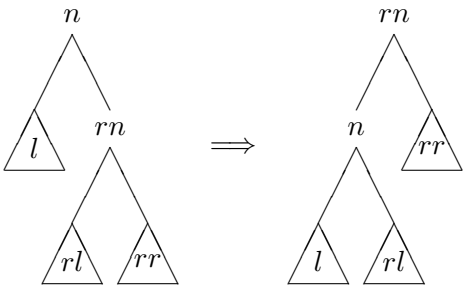

Figure 2: *l_rot*

tree is perfectly balanced, or heavier on the right, or heavier on the left. The necessary rotations are illustrated in Fig. 1-4.

1.1. AVL tree insertion

We explain insertion of a node into an AVL tree in order to motivate the use of the rotation functions. Suppose we insert a node x into a (balanced ordered) tree $Node\, n\, l\, r$. If $x = n$, then x should not be inserted at all since the property of being ordered requires that the tree contains no duplicates. If $x < n$, we must insert x into l (to maintain the ordering). Let l' be the tree obtained by inserting x into l, and assume that it is balanced and ordered. As an intermediate result, we have the tree $Node\, n\, l'\, r$. It is ordered, but it might not be balanced.

In fact, it might be the case that $height\ l = height\ r + 1$ and $height\ l' = height\ l + 1$. Then $height\ l' = height\ r + 2$ and so $Node\ n\ l'\ r$ is not balanced.

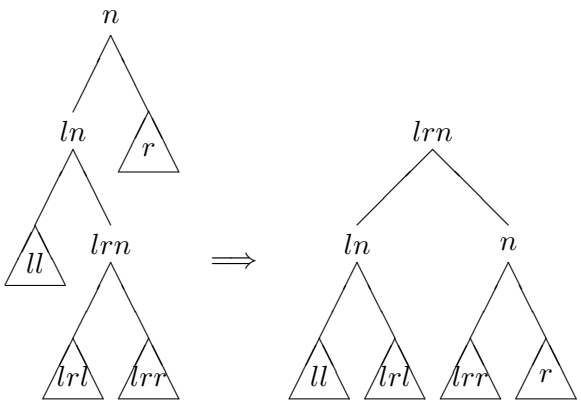

Figure 3: *lr_rot*

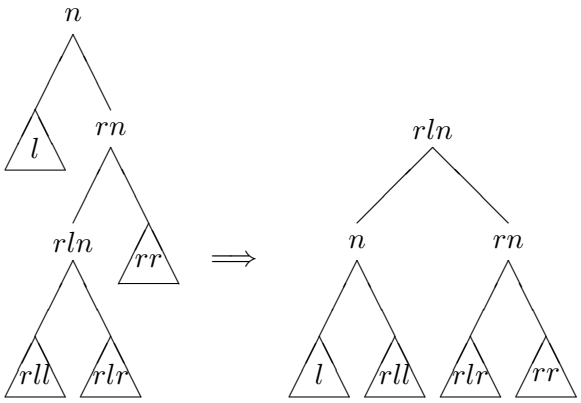

Figure 4: *rl_rot*

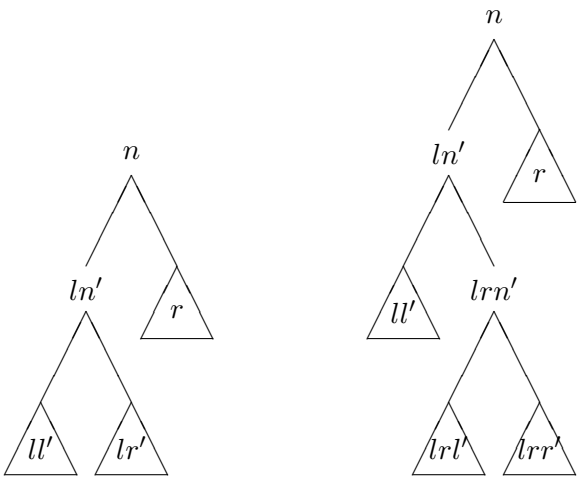

Figure 5: Too heavy on left

Note that in all other cases, *Node* $n\, l'\, r$ is balanced.

So suppose that *height* l' = *height* $r + 2$. Then *Node* $n\, l'\, r$ looks as shown in the first picture of Figure 5, where either *height* ll' = *height* $r + 1$ or *height* lr' = *height* $r + 1$[1]. The tree is too heavy on the left, and rotation must rectify this. We distinguish two cases:

bal l' = *Right*. Since l' is balanced, this means that *height* lr' = *height* $r + 1$ and *height* ll' = *height* r. So, since lr' has height > 0, it follows that *Node* $n\, l'\, r$ actually looks as shown in the second picture of Figure 5, where both lrl' and lrr' have height *height* r or *height* $r - 1$. Since all three trees ll', lrl', lrr' have the same height as r or one less, it follows that *lr_rot* produces a balanced tree (see Figure 3).

bal $l' \neq$ *Right*. In this case, *height* ll' = *height* $r + 1$, and, since l' is balanced, *height* lr' = *height* $r + 1$ or *height* lr' = *height* r. One can easily see that *r_rot* produces a balanced tree (see Figure 1).

1.2. Efficient AVL trees

In the sequel, we discuss more efficient implementations of AVL-tree's. New function definitions and enriched data types were given; at the end, lemmas

[1] Never both actually, but this is not needed in the proofs.

were proven that reveal the exact relationship between the new function versions processing new data to the old ones.

The overall scheme is also well-known as a *data refinement*.

The first inefficiency we noticed is that isin traverses the entire tree, which is unnecessary in case the tree is ordered. Note that for verification purposes, the more general but inefficient version is still sometimes in-dispensary.

Another inefficiency we noted is related to bal, which calls height for any node during insertion at the insertion path. A solution here is to store the result of bal as an additional attribute in an extended version of the AVL tree. As a consequence, this attribute must be kept consistent during operations on the enriched tree.

2. More on Isabelle

2.1. Some non-elementary constructs of ISAR

In the previous exercise, we have presented a core-language of ISAR, providing constructs such as **note** for binding (parts of) a current proof state as *fact* to a name that can be referenced later, or the **let** for introducing meta-variables as abbreviations of terms, which can also occur in propositions, substitutions or other pattern-match constructs such as (is <pattern>).

On top of this, we will now introduce a number of short-cuts that allow for an implicit management of facts and meta-variables.

also ≡ **note** calculation = **this** *(∗ initial case ∗)*
also ≡ **note** calculation = r ∘ (calculation @ **this**)
(∗ for some rule r out of some given set of transitivity rules ∗)

finally ≡ **also from** calculation
moreover ≡ **note** calculation = calculation @ **this**
ultimately ≡ **moreover from** calculation
then ≡ **from this**
thus ≡ hence ≡ then **show**
with <facts> ≡ **from** <facts> **this**

also
moreover
ultimately
then
thus
with

Here, calculation is a standard name for a list of facts, @ the concatenation on them, ∘ the forward resolution.

Similar to calculation , there is a generic name "..." which refers to the right-hand side of the most recent explicit fact statement. This allows to represent calculational sequences as follows:

```
have "x1 = x2" <proof>
also have "... = x3" <proof>
also have "... = x4" <proof>
finally have x1 = x4 .
```

Note that the "." at the very end is again an abbreviation for **by**(**this**).

One of the more trickier constructs of ISAR is the case distinction construct, which works as well for case-splits as for inductions. For a current proof state, goal by goal, it allows for creating sub-proofs referenced by names. The details of this construction are quite involved (see Nipkow's Paper "Structured Proofs in ISAR/HOL" for details), here we give just an example:

```
lemma "length(tl xs) = length xs −1
proof (cases xs)
  case Nil          thus ?case by simp
next
  case (Cons y ys) thus ?case by simp
qed
```

3. Exercises

Get the template theory `http://www.infsec.ethz.ch/education/permanent/csmr/material/HOL_AVL_tmpl.thy` and complete it.

3.1. Exercise 46

Define the function *insert* :: "'a::order ⇒ 'a tree <\Rightarrow> 'a tree". This involves the definition of:

- height, which computes the maximal number of nodes on a path from the root to a leaf,
- is_ord , which decides that for each node labeled n, all node labels in the left subtree are smaller, and all labels in the right subtree are greater,
- is_bal , which decides that it is either a leaf or a node with balanced subtrees where the height differs at most by one,
- is_in_eff which should provide a $O(lnn)$ implementation for ordered trees,
- the elementary rotation operations l_rot and lr_rot (analogously to the given functions r_rot and rl_rot),

- the balancing operation r_bal (analogously to the given functions l_bal),
- and finally the *insert* function.

Of course, your definitions should allow to prove the properties in the subsequent lemmas.

Hint: Use the general well-founded recursion mechanism:

```
recdef f "<wf_order"
  "f pat_1 = ... "
  ...
  "f pat_n = ... "
```

supporting pattern matching as in SML or Haskell whenever necessary.

3.2. Exercise 47

Prove two of the following properties of your programs:

1.

```
lemma is_in_insert :
" is_in y ( insert x t) = (y=x ∨ is_in y t)"
```

2.

```
lemma is_in_eff_correct [rule_format (no_asm)]:
" is_ord t _ ( is_in k t = is_in_eff k t)"
```

3.

```
lemma is_ord_insert :
" is_ord t d is_ord ( insert (x ::' a :: linorder ) t)"
```

Hint: After giving the definitions in Exercise 46, the commented proof scripts should work again. Try to prove analogous cases and to flush out the **sorry**'s.

3.3. Exercise 48 (*optional, tricky*)

A data refinement is provided by the new tree structure:

datatype 'a etree = EET | EMKT bal 'a "'a etree" "'a etree"

where the balancing information is directly stored in the tree.

1. Define a recursive definition of insertE on etree's that avoids the re-computation of height.
2. Speculate: What should be the crucial properties of this definition ? (state lemmas with sorry)!
3. Speculate: What could be a possible proof plan (state lemmas with sorry)?

A. Encoding AVL trees in in Isabelle (skeleton)

```
1   theory AVL = Main:
2
3   datatype 'a tree = ET | MKT 'a "'a tree" "'a tree"
4
5   consts
6     height :: "'a tree ⇒ nat"
7     is_in  :: "'a ⇒ 'a tree ⇒ bool"
8     is_ord :: "('a::order) tree ⇒ bool"
9     is_bal :: "'a tree ⇒ bool"
10
11
12  primrec
13    "is_in k ET = False"
14    "is_in k (MKT n l r) = (k=n ∨is_in k l ∨ is_in k r)"
15
16
17  (* ********************************************** *)
18  (* Define height, is_ord, is_bal, is_in_eff here ... *)
19  (* ********************************************** *)
20
21  datatype bal = Just | Left | Right
22
23  constdefs
24    bal :: "'a tree ⇒ bal"
25    "bal t ≡case t of ET          ⇒ Just
26                    | (MKT n l r) ⇒if height l = height r then Just
27                                    else if height l < height r
28                                         then Right else Left"
29
30  consts
31    r_rot  :: "'a × 'a tree × 'a tree ⇒ 'a tree"
32    l_rot  :: "'a × 'a tree × 'a tree ⇒ 'a tree"
33    lr_rot :: "'a × 'a tree × 'a tree ⇒ 'a tree"
34    rl_rot :: "'a × 'a tree × 'a tree ⇒ 'a tree"
35
36
37  recdef r_rot "{}"
38    "r_rot (n, MKT ln ll lr, r) = MKT ln ll (MKT n lr r)"
39
40  recdef rl_rot "{}"
41    "rl_rot (n, l, MKT rn (MKT rln rll rlr) rr) =
42                MKT rln (MKT n l rll) (MKT rn rlr rr)"
43
44
```

```
45  (* ************************************************ *)
46  (* Define the analogous functions  l_rot  and lr_rot  here *)
47  (* ************************************************ *)
48
49
50  constdefs
51    l_bal  ::  "'a ⇒ 'a tree ⇒ 'a tree ⇒ 'a tree"
52    "l_bal  n l  r ≡ if  bal  l  =  Right
53                     then lr_rot  (n,  l,  r)
54                     else  r_rot  (n,  l,  r)"
55
56
57  (* ************************************************ *)
58  (* Define the analogous function  rbal  here.                *)
59  (* ************************************************ *)
60
61
62  (* ************************************************ *)
63  (* Define the  insert  function for  'a::order on 'a tree .  *)
64  (* ************************************************ *)
65  consts
66     insert   ::  "'a::order ⇒ 'a tree  ⇒ 'a tree"
67
68  subsection "is-bal"
69
70  declare Let_def [simp]
71
72  lemma is_bal_lr_rot:
73  "⟦ height  l  =  Suc(Suc(height r));
74     bal  l  =  Right;  is_bal  l;  is_bal  r ⟧
75   ⟹ is_bal  ( lr_rot  (n,  l,  r))"
76  sorry
77
78
79  lemma is_bal_r_rot:
80  "⟦ height  l  =  Suc(Suc(height r));
81     bal  l  ≠ Right;  is_bal  l;  is_bal  r ⟧
82   ⟹ is_bal  ( r_rot  (n,  l,  r))"
83  sorry
84
85
86  lemma is_bal_rl_rot:
87  "⟦ height  r  =  Suc(Suc(height l));
88     bal  r  =  Left;  is_bal  l;  is_bal  r ⟧
89   ⟹ is_bal  ( rl_rot  (n,  l,  r))"
90  sorry
91
92
93  lemma is_bal_l_rot:
94  "⟦ height  r  =  Suc(Suc(height l));  bal  r  ≠ Left;  is_bal  l;  is_bal  r ⟧
95     ⟹ is_bal  ( l_rot  (n,  l,  r))"
96  (*
97  apply  (unfold  bal_def )
98  apply  (cases  r)
99   apply  simp
100 apply  (simp add: max_def  split :  split_if_asm )
101 done
102  *)
103 sorry
104
105 text {* Lemmas about height after rotation *}
106
107 lemma height_lr_rot:
108 "⟦ bal  l  =  Right; height  l  =  Suc(Suc(height r)) ⟧
109  ⟹ Suc(height ( lr_rot  (n,  l,  r)))  =  height (MKT n l r) "
110 (*
111 apply  (unfold  bal_def )
112 apply  (cases  l)
113  apply  simp
114 apply  (rename_tac t1  t2)
115 apply  ( case_tac  t2)
116  apply  simp
117 apply  (simp add: max_def  split :  split_if_asm )
118 done
119 *)
120 sorry
```

```
121
122 lemma height_r_rot:
123 "⟦ height l = Suc(Suc(height r)); bal l ≠ Right ⟧
124   ⟹ Suc(height (r_rot (n, l, r))) = height (MKT n l r) ∨
125           height (r_rot (n, l, r)) = height (MKT n l r)"
126 sorry
127
128
129 lemma height_l_bal:
130 "height l = Suc(Suc(height r))
131   ⟹ Suc(height (l_bal n l r)) = height (MKT n l r) |
132           height (l_bal n l r) = height (MKT n l r)"
133 sorry
134
135
136 lemma height_rl_rot [rule_format (no_asm)]:
137 "height r = Suc(Suc(height l)) ⟶ bal r = Left
138   ⟶ Suc(height (rl_rot (n, l, r))) = height (MKT n l r)"
139 sorry
140
141 lemma height_l_rot [rule_format (no_asm)]:
142 "height r = Suc(Suc(height l)) ⟶ bal r ≠ Left
143  ⟶ Suc(height (l_rot (n, l, r))) = height (MKT n l r) ∨
144           height (l_rot (n, l, r)) = height (MKT n l r)"
145 sorry
146
147
148
149 lemma height_r_bal:
150 "height r = Suc(Suc(height l))
151   ⟹ Suc(height (r_bal n l r)) = height (MKT n l r) ∨
152           height (r_bal n l r) = height (MKT n l r)"
153 (*
154 apply (unfold r_bal_def)
155 apply (cases "bal r = Left")
156  apply (fastsimp dest: height_rl_rot)
157 apply (fastsimp dest: height_l_rot)
158 done
159 *)
160 sorry
161
162
163 lemma height_insert [rule_format (no_asm)]:
164 "is_bal t
165   ⟶ height (insert x t) = height t ∨ height (insert x t) = Suc(height t)"
166 sorry
167 (*
168 apply (induct_tac "t")
169  apply simp
170 apply (rename_tac n t1 t2)
171 apply (case_tac "x=n")
172  apply simp
173 apply (case_tac "x<n")
174  apply (case_tac "height (insert x t1) = Suc (Suc (height t2))")
175   apply (frule_tac n = n in height_l_bal)
176   apply (simp add: max_def)
177   apply fastsimp
178  apply (simp add: max_def)
179  apply fastsimp
180 apply (case_tac "height (insert x t2) = Suc (Suc (height t1))")
181  apply (frule_tac n = n in height_r_bal)
182  apply (fastsimp simp add: max_def)
183 apply (simp add: max_def)
184 apply fastsimp
185 done
186 *)
187
188 lemma is_bal_insert_left:
189 "⟦height (insert x l) ≠ Suc(Suc(height r));
190   is_bal (insert x l); is_bal (MKT n l r)⟧
191  ⟹ is_bal (MKT n (insert x l) r)"
192 sorry
193
194
195 lemma is_bal_insert_right:
196 "⟦ height (insert x r) ≠ Suc(Suc(height l));
```

```
197       is_bal  ( insert  x r );  is_bal  (MKT n l r) ⟧
198    ⟹ is_bal  (MKT n l (insert x r))"
199   sorry
200
201   lemma is_bal_insert [rule_format (no_asm)]:
202   " is_bal  t  ⟶ is_bal ( insert  x t)"
203   sorry
204
205   subsection " is−in"
206
207   lemma is_in_lr_rot:
208   "⟦ height  l = Suc(Suc(height r));  bal  l = Right ⟧
209    ⟹ is_in  x ( lr_rot  (n,  l,  r)) = is_in  x (MKT n l r)"
210   sorry
211   (*
212   apply (unfold  bal_def )
213   apply (cases  l)
214    apply simp
215   apply (rename_tac t1 t2)
216   apply ( case_tac  t2)
217    apply simp
218   apply fastsimp
219   done
220    *)
221
222   lemma is_in_r_rot:
223   "⟦ height  l = Suc(Suc(height r));  bal  l ≠ Right ⟧
224      ⟹ is_in  x ( r_rot  (n,  l,  r)) = is_in  x (MKT n l r)"
225   sorry
226   (*
227   apply (unfold  bal_def )
228   apply (cases  l)
229    apply simp
230   apply fastsimp
231   done
232    *)
233
234   lemma is_in_rl_rot:
235   "⟦ height  r = Suc(Suc(height l));  bal  r = Left ⟧
236    ⟹ is_in  x ( rl_rot  (n,  l,  r)) = is_in  x (MKT n l r)"
237   sorry
238   (*
239   apply (unfold  bal_def )
240   apply (cases  r)
241    apply simp
242   apply (rename_tac t1 t2)
243   apply ( case_tac  t1)
244    apply (simp add: max_def  le_def )
245   apply fastsimp
246   done
247    *)
248
249   lemma is_in_l_rot:
250   "⟦ height  r = Suc(Suc(height l));  bal  r ≠ Left ⟧
251      ⟹ is_in  x ( l_rot  (n,  l,  r)) = is_in  x (MKT n l r)"
252   sorry
253   (*
254   apply (unfold  bal_def )
255   apply (cases  r)
256    apply simp
257   apply fastsimp
258   done
259    *)
260
261   lemma is_in_insert:
262   " is_in  y ( insert  x t) = (y=x ∨is_in  y t)"
263   sorry
264
265   lemma is_in_ord_l [rule_format (no_asm)]:
266   " is_ord  (MKT n l r) ⟶x < n ⟶is_in  x (MKT n l r) ⟶is_in  x l"
267   sorry
268
269   lemma is_in_ord_r [rule_format (no_asm)]:
270   " is_ord  (MKT n l r) ⟶n < x ⟶is_in  x (MKT n l r) ⟶is_in  x r"
271   sorry
272
```

```
subsection "is-in-eff"

lemma is_in_eff_correct [rule_format (no_asm)]:
"is_ord t ⟶ (is_in k t = is_in_eff k t)"
sorry


subsection "is-ord"

lemma is_ord_lr_rot [rule_format (no_asm)]:
"⟦ height l = Suc(Suc(height r));
   bal l = Right; is_ord (MKT n l r) ⟧
 ⟹ is_ord (lr_rot (n, l, r))"
sorry
(*
apply (unfold bal_def)
apply (cases l)
 apply simp
apply (rename_tac t1 t2)
apply (case_tac t2)
 apply simp
apply simp
apply (blast intro: order_less_trans)
done
 *)

lemma is_ord_r_rot:
"⟦ height l = Suc(Suc(height r));
   bal l ≠ Right; is_ord (MKT n l r) ⟧
 ⟹ is_ord (r_rot (n, l, r))"
sorry
(*
apply (unfold bal_def)
apply (cases l)
apply (simp (no_asm_simp))
apply (auto intro: order_less_trans)
done
 *)

lemma is_ord_rl_rot:
"⟦ height r = Suc(Suc(height l));
   bal r = Left; is_ord (MKT n l r) ⟧
 ⟹ is_ord (rl_rot (n, l, r))"
sorry
(*
apply (unfold bal_def)
apply (cases r)
 apply simp
apply (rename_tac t1 t2)
apply (case_tac t1)
 apply (simp add: le_def)
apply simp
apply (blast intro: order_less_trans)
done
*)

lemma is_ord_l_rot:
"⟦ height r = Suc(Suc(height l)); bal r ≠ Left; is_ord (MKT n l r) ⟧
  ⟹ is_ord (l_rot (n, l, r))"
sorry
(*
apply (unfold bal_def)
apply (cases r)
 apply simp
apply simp
apply (blast intro: order_less_trans)
done
*)


(* insert operation presreves is_ord property *)

lemma is_ord_insert:
"is_ord t ⟹ is_ord(insert (x::'a::linorder) t)"
sorry
```

```
subsection "An extended tree datatype with labels for the balancing information"

datatype 'a etree = EET | EMKT bal 'a "'a etree" "'a etree"


text {* Pruning, i.e. throwing away the balancing labels : *}
consts
  strip :: "'a etree ⇒ 'a tree"
primrec
  "strip EET = ET"
  "strip (EMKT b n l r) =
                          MKT n (strip l) (strip r)"

text {* Test if the balancing arguments are correct : *}
consts
   correct_labelled :: "'a etree ⇒ bool"
primrec
  "correct_labelled EET = True"
  "correct_labelled (EMKT b n l r) =
                  (b = bal (MKT n (strip l) (strip r))
                   ∧ correct_labelled l
                   ∧ correct_labelled r)"

text {* Add correct balancing labels : *}
consts
  label :: "'a tree ⇒ 'a etree"
primrec
 "label ET = EET"
 "label (MKT n l r) = EMKT (bal (MKT n l r)) n (label l)
                                                (label r)"

lemma correct_strip:
"correct_labelled (EMKT b n l r) ⟶(bal (strip (EMKT b n l r)) = b)"
apply (simp (no_asm_simp) add: bal_def)
done

subsection "Reversing of strip and label"

lemma prune_label: "strip (label t) = t"
apply (induct_tac "t")
apply (simp (no_asm))
apply (simp (no_asm))
apply (erule_tac conjI)
apply assumption
done

lemma label_prune: "correct_labelled t ⟹ label (strip t) = t"
apply (induct t)
apply auto
done

end
```