A General Method for the Proof of Theorems on Tail-recursive Functions

Pasquale Noce (pasquale dot noce at arjowiggins-it dot com)

May 27, 2015

Abstract

Tail-recursive function definitions are sometimes more straightforward than alternatives, but proving theorems on them may be roundabout because of the peculiar form of the resulting recursion induction rules.

This paper describes a proof method that provides a general solution to this problem by means of suitable invariants over inductive sets, and illustrates the application of such method by examining two case studies.

Contents

1 Method rationale 2
2 Method summary 6
3 Case study 1 8
   3.1 Step 1 10
   3.2 Step 2 10
   3.3 Step 3 10
   3.4 Step 4 11
   3.5 Step 5 13
   3.6 Step 6 14
   3.7 Step 7 14
   3.8 Step 8 14
   3.9 Step 9 14
   3.10 Step 10 17
4 Case study 2 17
   4.1 Step 1 20
   4.2 Step 2 21
   4.3 Step 3 21
   4.4 Step 4 22
1 Method rationale

Tail-recursive function definitions are sometimes more intuitive and straightforward than alternatives, and this alone would be enough to make them preferable in such cases for the mere purposes of functional programming. However, proving theorems about them with a formal proof assistant like Isabelle may be roundabout because of the peculiar form of the resulting recursion induction rules.

Let:

- \textit{f-naive} be a tail-recursive function of type \( a_1 \Rightarrow ... \Rightarrow a_n \Rightarrow b \).
- \( a \) be an \( n \)-tuple of values of types \( a_1, ..., a_n \) such that the computation of \textit{f-naive} \( a \), say outputting value \( b \), involves at least one recursive call – which is what happens in general for significant inputs (e.g. those complying with initial conditions for accumulator arguments), as otherwise a non-recursive function definition would be sufficient.

- \( a_1, ..., a_m \) be the sequence of the intermediate \( n \)-tuples of values of types \( a_1, ..., a_n \) arising from the computation of \textit{f-naive} \( a \).
- \( f\text{-naive} \ X_1 = f\text{-naive} \ X'_1, ..., f\text{-naive} \ X_m = f\text{-naive} \ X'_m, f\text{-naive} \ X = Y \) be the sequence (possibly with repetitions) of the equations involved in the computation of \textit{f-naive} \( a \) – which implies that, putting \( a_0 = a \), they are satisfied for \((X_1, X'_1) = (a_0, a_1), ..., (X_m, X'_m) = (a_{m-1}, a_m), (X, Y) = (a_m, b)\), respectively.

That being stated, suppose that theorem \( P \ (f\text{-naive} \ a) \) has to be proven. If recursion induction is applied to such goal, for each \( i \in \{1..m\} \), the recursive equation \( f\text{-naive} \ X_i = f\text{-naive} \ X'_i \) gives rise to subgoal \( P \ (f\text{-naive} \ X'_i) \implies P \ (f\text{-naive} \ X_i) \), trivially discharged by simplification. On the contrary, the non-recursive equation \( f\text{-naive} \ X = Y \) brings about the generation of subgoal \( P \ (f\text{-naive} \ X) \), which is intractable unless it trivially follows from either the equation or the form of pattern \( X \).

Indeed, in non-trivial cases such as the case studies examined in this paper, this formula even fails to be a theorem, thus being hopeless as a goal, since it is false for some values of its variables. The reason for this is that non-trivial properties of the output of tail-recursive functions depend on the
input as well as on the whole recursive call pipeline leading from the input to the output, and all of this information corresponds to missing necessary assumptions in subgoal $P\ (f\text{-naive } X)$.

Therefore, for a non-trivial theorem $P\ (f\text{-naive } a)$, recursion induction is rather applicable to some true conditional statement $f\text{-inv } x \rightarrow P\ (f\text{-naive } x)$ complying with both of the following requirements:

- subgoal $f\text{-inv } X \rightarrow P\ (f\text{-naive } X)$ arising from equation $f\text{-naive } X = Y$ be tractable, and

- formula $f\text{-inv } a$ can be shown to be true, so that theorem $P\ (f\text{-naive } a)$ can be inferred from conditional $f\text{-inv } a \rightarrow P\ (f\text{-naive } a)$ by modus ponens.

Observe that the antecedent of the conditional may not have the form $f\text{-inv } (f\text{-naive } x)$. Otherwise, the latter requirement would ask for proving formula $f\text{-inv } (f\text{-naive } a)$, which would be at least as hard to prove as formula $P\ (f\text{-naive } a)$ being the former a sufficient condition for the latter. Hence, the same problem as that originating from the proof of formula $P\ (f\text{-naive } a)$ would have to be solved again, which would give rise to a regressio ad infinitum.

The latter requirement entails that formula $f\text{-inv } a_0$ holds. Moreover, for each $i \in \{1..m\}$, in the proof of conditional $f\text{-inv } x \rightarrow P\ (f\text{-naive } x)$ by recursion induction, the recursive equation $f\text{-naive } X_i = f\text{-naive } X'_i$ brings about the generation of subgoal $f\text{-inv } X'_i \rightarrow P\ (f\text{-naive } X'_i) \implies f\text{-inv } X_i \rightarrow P\ (f\text{-naive } X_i)$. Assuming that formula $f\text{-inv } a_{i-1}$ holds, it turns out that the conclusion antecedent $f\text{-inv } X_i$ may not be shown to be false, as $n$-tuple $a_{i-1}$ matches pattern $X_i$; thus, the conclusion consequent $P\ (f\text{-naive } X_i)$ has to be proven.

In non-trivial cases, this requires that the assumption antecedent $f\text{-inv } X'_i$ be derived from the conclusion antecedent $f\text{-inv } X_i$ used as a further assumption, so that the assumption consequent $P\ (f\text{-naive } X'_i)$ – matching $P\ (f\text{-naive } X_i)$ by virtue of equation $f\text{-naive } X_i = f\text{-naive } X'_i$ – can be proven by modus ponens. This in turn requires that $f\text{-inv } X_i$ imply $f\text{-inv } X'_i$, i.e. that $f\text{-inv } x_i$ imply $f\text{-inv } x'_i$ for any pair of $n$-tuples $x_i, x'_i$ matching patterns $X_i, X'_i$ with respect to the same value assignment. But such are $n$-tuples $a_{i-1}, a_i$ as they solve equation $f\text{-naive } X_i = f\text{-naive } X'_i$, so that the supposed truth of $f\text{-inv } a_{i-1}$ entails that of $f\text{-inv } a_i$.

Hence, by induction, all of formulae $f\text{-inv } a, f\text{-inv } a_1, ..., f\text{-inv } a_m$ turn out to be true. On the other hand, the former requirement is verified if either the antecedent $f\text{-inv } X$ can be shown to be false, which would entail its falsity for any $n$-tuple matching pattern $X$, or else the consequent $P\ (f\text{-naive } X)$ can be shown to be true using the antecedent as an assumption. Since formula $f\text{-inv } a_m$ is true and $n$-tuple $a_m$ matches pattern $X$, the case that actually occurs is the second one.
Thus, the former requirement is equivalent to asking for an introduction rule to be proven – in fact, a conditional with a contradiction as antecedent may not be used as an introduction rule – having the form \( f\text{-inv } X \Rightarrow P \) (\( f\text{-naive } X \)), or rather \([f\text{-inv } x; f\text{-form } x] \Rightarrow P \) (\( f\text{-naive } x \)) for a suitable predicate \( f\text{-form} \) satisfied by any \( n \)-tuple matching pattern \( X \). In the degenerate case in which the rule can be shown to be true for \( f\text{-form} \) \( = (\lambda x. \text{True}) \), it admits to be put into the simpler equivalent form \( f\text{-inv } x \Rightarrow P \) (\( f\text{-naive } x \)).

An even more important consequence of the previous argument is that in non-trivial cases, the task of proving conditional \( f\text{-inv } x \rightarrow P \) (\( f\text{-naive } x \)) by recursion induction requires that \( f\text{-inv } X' \) be derived from \( f\text{-inv } X \) for each recursive equation \( f\text{-naive } X_i = f\text{-naive } X_i' \), where \( i \in \{1..m\} \).

Let:

- \( 'a \) be the Cartesian product of types \( 'a_1, ..., 'a_n \).
- \( f\text{-set} \) be the inductive set of type \( 'a \Rightarrow 'a \) set defined by introduction rules \( x \in f\text{-set } x, X_1 \in f\text{-set } x \Rightarrow X'_1 \in f\text{-set } x, ..., X_m \in f\text{-set } x \Rightarrow X'_m \in f\text{-set } x \) – where patterns \( X_1, X'_1, ..., X_m, X'_m \) are now viewed as values of type \( 'a \).

Then, the problem of discharging the above proof obligation on predicate \( f\text{-inv} \) is at least as hard as that of proving by rule induction introduction rule \([y \in f\text{-set } x; f\text{-inv } x] \Rightarrow f\text{-inv } y \) – which states that for any \( x \) such that \( f\text{-inv } x \) is true, \( f\text{-inv} \) is an invariant over inductive set \( f\text{-set } x \), i.e. \( f\text{-inv } y \) is true for each \( y \in f\text{-set } x \).

In fact, the application of rule induction to this goal generates subgoals \( f\text{-inv } x \Rightarrow f\text{-inv } x, [X_1 \in f\text{-set } x; f\text{-inv } X_1; f\text{-inv } x] \Rightarrow f\text{-inv } X'_1, ..., [X_m \in f\text{-set } x; f\text{-inv } X_m; f\text{-inv } x] \Rightarrow f\text{-inv } X'_m \); the first is trivial, and such would also be the other ones if rules \( f\text{-inv } X_1 \Rightarrow f\text{-inv } X'_1, ..., f\text{-inv } X_m \Rightarrow f\text{-inv } X'_m \) were available.

Furthermore, suppose that the above invariance property of predicate \( f\text{-inv} \) have been proven; then, the proof of conditional \( f\text{-inv } x \rightarrow P \) (\( f\text{-naive } x \)) by recursion induction can be made unnecessary by slightly refining the definition of function \( f\text{-naive} \), as shown in the continuation.

Let \( f\text{-aux} \) be the tail-recursive function of type \( 'a \Rightarrow 'a \) whose definition is obtained from that of \( f\text{-naive} \) by treating as fixed points the patterns to which non-recursive equations apply as well as those to which no equation applies, if any – i.e. by replacing recursive equation \( f\text{-naive } X_i = f\text{-naive } X_i' \) with \( f\text{-aux } X_i = f\text{-aux } X_i' \) for each \( i \in \{1..m\} \) and non-recursive equation \( f\text{-naive } X = Y \) with \( f\text{-aux } X = Y \).

Then, define function \( f \) by means of a non-recursive equation \( f x = f\text{-out } (f\text{-aux } (f\text{-in } x)) \), where:

- \( f\text{-in} \) is a function of type \( 'a \Rightarrow 'a \), for a suitable type \( 'a \) whose range contains all the significant inputs of function \( f\text{-naive} \).
• \( f\text{-out} \) is a function of type \( 'a \Rightarrow 'b \) mapping the outputs of \( f\text{-aux} \) to those of \( f\text{-naive} \), i.e. the values of type \( 'a \) matching pattern \( X \) to those of type \( 'b \) matching pattern \( Y \) with respect to the same value assignment.

The definitions of functions \( f\text{-aux} \) and \( f\text{-out} \) entail that equation \( f\text{-naive} \ x = f\text{-out} \ (f\text{-aux} \ x) \) holds for any \( x \). Particularly, \( f\text{-naive} \ a = f\text{-out} \ (f\text{-aux} \ a) \); thus, being \( a' \) an inverse image of \( a \) under \( f\text{-in} \), viz. \( a = f\text{-in} \ a' \), it follows that \( f\text{-naive} \ a = f \ a' \). As a result, theorem \( P \ (f\text{-naive} \ a) \) may be rewritten as \( P \ (f \ a') \).

For any \( x \), \( f\text{-set} \ x \) is precisely the set of the values recursively input to function \( f\text{-aux} \) in the computation of \( f\text{-aux} \ x \), including \( x \) itself, and it can easily be ascertained that \( f\text{-aux} \ x \) is such a value. In fact, the equation invoked last in the computation of \( f\text{-aux} \ x \) must be a non-recursive one, so that it has the form \( f\text{-aux} \ X = X \), since all non-recursive equations in the definition of \( f\text{-aux} \) apply to fixed points. Thus, being \( f\text{-aux} \ x \) the output of the computation, the right-hand side of the equation, i.e. the pattern \( X \) also input to function \( f\text{-aux} \) in the left-hand side, is instantiated to value \( f\text{-aux} \ x \).

Therefore, \( f\text{-aux} \ x \in f\text{-set} \ x \) for any \( x \). Observe that the argument rests on the assumption that whatever \( x \) is given, a sequence of equations leading from \( x \) to \( f\text{-aux} \ x \) be actually available – and what is more, nothing significant could be proven on \( f\text{-aux} \ x \) for any \( x \) for which its value were undefined, and then arbitrary. The trick of making the definition of \( f\text{-aux} \) total by adding equations for the patterns not covered in the definition of \( f\text{-naive} \), if any, guarantees that this assumption be satisfied.

An additional consequence of the previous argument is that \( f\text{-aux} \ (f\text{-aux} \ x) = f\text{-aux} \ x \) for any \( x \), i.e. function \( f\text{-aux} \) is idempotent. If introduction rule \( [f\text{-inv} \ x; f\text{-form} \ x] \implies P \ (f\text{-naive} \ x) \) is rewritten by applying equation \( f\text{-naive} \ x = f\text{-out} \ (f\text{-aux} \ x) \), instantiating free variable \( x \) to \( f\text{-aux} \ x \), and then applying the idempotence of function \( f\text{-aux} \), the result is formula \( [f\text{-inv} \ (f\text{-aux} \ x); f\text{-form} \ (f\text{-aux} \ x)] \implies P \ (f\text{-out} \ (f\text{-aux} \ x)) \), which is nothing but an instantiation of introduction rule \( [f\text{-inv} \ x; f\text{-form} \ x] \implies P \ (f\text{-out} \ x) \).

Observe that this rule is just a refinement of a rule whose proof is required for proving conditional \( f\text{-inv} \ x \longrightarrow P \ (f\text{-naive} \ x) \) by recursion induction, so that it does not give rise to any additional proof obligation. Moreover, it contains neither function \( f\text{-naive} \) nor \( f\text{-aux} \), thus its proof does not require recursion induction with respect to the corresponding induction rules.

The instantiation of such refined introduction rule with value \( f\text{-aux} \ a \) is \( [f\text{-inv} \ (f\text{-aux} \ a); f\text{-form} \ (f\text{-aux} \ a)] \implies P \ (f\text{-out} \ (f\text{-aux} \ a)) \), which by virtue of equality \( a = f\text{-in} \ a' \) and the definition of function \( f \) is equivalent to formula \( [f\text{-inv} \ (f\text{-aux} \ a); f\text{-form} \ (f\text{-aux} \ a)] \implies P \ (f \ a') \). Therefore, the rule is sufficient to prove theorem \( P \ (f \ a') \) – hence making unnecessary the proof of conditional \( f\text{-inv} \ x \longrightarrow P \ (f\text{-naive} \ x) \) by recursion induction, as
mentioned previously – provided the instantiated assumptions $f$-inv ($f$-aux $a$), $f$-form ($f$-aux $a$) can be shown to be true.

This actually is the case: the former assumption can be derived from formulae $f$-aux $a \in f$-set $a$, $f$-inv $a$ and the invariance of predicate $f$-inv over $f$-set $a$, while the latter can be proven by recursion induction, as by construction goal $f$-form $X$ is trivial for any pattern $X$ to which some non-recursive equation in the definition of function $f$-naive applies. If further non-recursive equations whose patterns do not satisfy predicate $f$-form have been added to the definition of $f$-aux to render it total, rule inversion can be applied to exclude that $f$-aux $a$ may match any of such patterns, again using formula $f$-aux $a \in f$-set $a$.

2 Method summary

The general method developed so far can be schematized as follows.

Let $f$-naive be a tail-recursive function of type $'a_1 \Rightarrow ... \Rightarrow 'a_n \Rightarrow 'b$, and $P$ ($f$-naive $a_1 ... a_n$) be a non-trivial theorem having to be proven on this function.

In order to accomplish such task, the following procedure shall be observed.

- **Step 1** — Refine the definition of $f$-naive into that of an auxiliary tail-recursive function $f$-aux of type $'a \Rightarrow 'a$, where $'a$ is a product or record type with types $'a_1, ..., 'a_n$ as components, by treating as fixed points the patterns to which non-recursive equations apply as well as those to which no equation applies, if any.

- **Step 2** — Define a function $f$ of type $'a' \Rightarrow 'b$ by means of a non-recursive equation $f x = f$-out ($f$-aux ($f$-in $x$)), where $f$-in is a function of type $'a' \Rightarrow 'a$ (possibly matching the identity function) whose range contains all the significant inputs of function $f$-naive, and $f$-out is a function of type $'a \Rightarrow 'b$ mapping the outputs of $f$-aux to those of $f$-naive.

Then, denoting with $a$ the value of type $'a$ with components $a_1, ..., a_n$, and with $a'$ an inverse image of $a$ under function $f$-in, the theorem to be proven takes the equivalent form $P$ ($f$ $a'$).

- **Step 3** — Let $f$-aux $X_1 = f$-aux $X'_1$, ..., $f$-aux $X_m = f$-aux $X'_m$ be the recursive equations in the definition of function $f$-aux.

Then, define an inductive set $f$-set of type $'a \Rightarrow 'a$ set with introduction rules $x \in f$-set $x$, $X_1 \in f$-set $x \Rightarrow X'_1 \in f$-set $x$, ..., $X_m \in f$-set $x \Rightarrow X'_m \in f$-set $x$.

If the right-hand side of some recursive equation contains conditionals in the form of if or case constructs, the corresponding introduction
rule can be split into as many rules as the possible mutually exclusive cases; each of such rules shall then provide for the related case as an additional assumption.

- **Step 4** — Prove lemma \( f\text{-aux } x \in f\text{-set } x \); a general inference scheme, independent of the specific function \( f\text{-aux} \), applies to this proof.
  First, prove lemma \( y \in f\text{-set } x \implies f\text{-set } y \subseteq f\text{-set } x \), which can easily be done by rule induction.
  Next, applying recursion induction to goal \( f\text{-aux } x \in f\text{-set } x \) and then simplifying, a subgoal \( X_i \in f\text{-set } X_i \) arises for each non-recursive equation \( f\text{-aux } X_i = X_i \); while a subgoal \( f\text{-aux } X'_j \in f\text{-set } X'_j \) arises for each recursive equation \( f\text{-aux } X_i = f\text{-aux } X'_j \).
  The former subgoals can be proven by introduction rule \( x \in f\text{-set } x \), the latter ones as follows: rule instantiations \( X_j \in f\text{-set } X_j \) and \( X_j \in f\text{-set } X_j \implies X'_j \in f\text{-set } X_j \) imply formula \( X'_j \in f\text{-set } X_j \); thus \( f\text{-set } X'_j \subseteq f\text{-set } X_j \) by the aforesaid lemma; from this and subgoal assumption \( f\text{-aux } X'_j \in f\text{-set } X'_j \), subgoal conclusion \( f\text{-aux } X'_j \in f\text{-set } X_j \) ensues.
  As regards recursive equations containing conditionals, the above steps have to be preceded by a case distinction, so as to obtain further assumptions sufficient for splitting such conditionals.

- **Step 5** — Define a predicate \( f\text{-inv} \) of type \( 'a \Rightarrow bool \) in such a way as to meet the proof obligations prescribed by the following steps.

- **Step 6** — Prove lemma \( f\text{-inv } a \).
  In case of failure, return to step 5 so as to suitably change the definition of predicate \( f\text{-inv} \).

- **Step 7** — Prove introduction rule \( f\text{-inv } x \Rightarrow P \ (f\text{-out } x) \), or rather \( [f\text{-inv } x; f\text{-form } x] \Rightarrow P \ (f\text{-out } x) \), where \( f\text{-form} \) is a suitable predicate of type \( 'a \Rightarrow bool \) satisfied by any pattern to which some non-recursive equation in the definition of function \( f\text{-naive} \) applies.
  In case of failure, return to step 5 so as to suitably change the definition of predicate \( f\text{-inv} \).

- **Step 8** — In case an introduction rule of the second form has been proven in step 7, prove lemma \( f\text{-form } (f\text{-aux } a) \) by recursion induction.
  If the definition of function \( f\text{-aux} \) resulting from step 1 contains additional non-recursive equations whose patterns do not satisfy predicate \( f\text{-form} \), rule inversion can be applied to exclude that \( f\text{-aux } a \) may match any of such patterns, using instantiation \( f\text{-aux } a \in f\text{-set } a \) of the lemma proven in step 4.
• **Step 9** — Prove by rule induction introduction rule \([y \in f\text{-}set~x; f\text{-}inv x] \implies f\text{-}inv y\), which states the invariance of predicate \(f\text{-}inv\) over inductive set \(f\text{-}set~x\) for any \(x\) satisfying \(f\text{-}inv\).

In case of failure, return to step 5 so as to suitably change the definition of predicate \(f\text{-}inv\).

Observe that the order in which the proof obligations related to predicate \(f\text{-}inv\) are distributed among steps 6 to 9 is ascending in the effort typically required to discharge them. The reason why this strategy is advisable is that in case one step fails, which forces to revise the definition of predicate \(f\text{-}inv\) and then also the proofs already worked out, such proofs will be the least demanding ones so as to minimize the effort required for their revision.

• **Step 10** — Prove theorem \(P(f\text{-}a')\) by means of the following inference scheme.

First, derive formula \(f\text{-}inv(f\text{-}aux~a)\) from introduction rule \([y \in f\text{-}set x; f\text{-}inv x] \implies f\text{-}inv y\) and formulae \(f\text{-}aux~a \in f\text{-}set~a, f\text{-}inv~a\).

Then, derive formula \(P(f\text{-}out~(f\text{-}aux~a))\) from either introduction rule \(f\text{-}inv~x \implies P(f\text{-}out~x)\) or \([f\text{-}inv~x; f\text{-}form~x] \implies P(f\text{-}out~x)\) and formulae \(f\text{-}inv~(f\text{-}aux~a), f\text{-}form~(f\text{-}aux~a)\) (in the latter case).

Finally, derive theorem \(P(f\text{-}a')\) from formulae \(P(f\text{-}out~(f\text{-}aux~a)), a = f\text{-}in~a'\) and the definition of function \(f\).

In the continuation, the application of this method is illustrated by analyzing two case studies drawn from an exercise comprised in Isabelle online course material; see [1]. The salient points of definitions and proofs are commented; for additional information see Isabelle documentation, particularly [5], [4], [3], and [2].

### 3 Case study 1

theory *CaseStudy1*

imports *Main*

begin

In the first case study, the problem will be examined of defining a function \(l\text{-}sort\) performing insertion sort on lists of elements of a linear order, and then proving the correctness of this definition, i.e. that the lists output by the function actually be sorted and contain as many occurrences of any value as the input lists.

Such function constitutes a paradigmatic example of a function admitting a straightforward tail-recursive definition. Here below is a naive one:

fun \(l\text{-}sort\text{-}naive\) :: \('a::linorder\ list \Rightarrow 'a\ list \Rightarrow 'a\ list\) where
\[ l\text{-}sort\text{-}naive (x \# xs) ys \] = \[ l\text{-}sort\text{-}naive xs \] (ys @ [x]) | \\
\[ l\text{-}sort\text{-}naive (x \# xs) ys (z \# zs) = (if x \leq z \text{ then } l\text{-}sort\text{-}naive xs \] (ys @ x \# z \# zs) | \\
\[ \text{else } l\text{-}sort\text{-}naive (x \# xs) (ys @ [z]) zs | \\
\[ l\text{-}sort\text{-}naive \] ys zs = zs

The first argument is deputed to contain the values still having to be inserted into the sorted list, accumulated in the third argument. For each of such values, the items of the sorted list are orderly moved into a temporary one (second argument) to search the insertion position. Once found, the sorted list is restored, the processed value is moved from the unsorted list to the sorted one, and another iteration of the loop is performed up to the exhaustion of the former list.

A further couple of functions are needed to express the aforesaid correctness properties of function \( l\text{-}sort\text{-}naive \):

\[
\text{fun } l\text{-}sorted :: \text{'}a::\text{linorder list } \Rightarrow \text{ bool where}\\
l\text{-}sorted (x \# x' \# xs) = (x \leq x' \land l\text{-}sorted (x' \# xs)) | \\
l\text{-}sorted - = \text{True}
\]

\[
\text{definition } l\text{-}count :: \text{'}a \Rightarrow \text{'}a \text{ list } \Rightarrow \text{ nat where}\\l\text{-}count x xs \equiv \text{length [x'←xs. x' = x]}
\]

Then, the target correctness theorems can be enunciated as follows:

\[ l\text{-}sorted (l\text{-}sort\text{-}naive xs \] []]) \\
\[ l\text{-}count x (l\text{-}sort\text{-}naive xs \] []]) = l\text{-}count x xs

Unfortunately, attempts to apply recursion induction to such goals turn out to be doomed, as can easily be ascertained by considering the former theorem:

\[
\text{theorem } l\text{-}sorted (l\text{-}sort\text{-}naive xs \] []]) \\
\text{proof (rule l\text{-}sort\text{-}naive.induct [of } \lambda xs \text{ ys zs. } l\text{-}sorted (l\text{-}sort\text{-}naive xs ys zs)],[
\text{simp-all del: l\text{-}sort\text{-}naive.simps(3))}
\]

Simplification deletes all the subgoals generated by recursive equations. However, the following subgoal arises from the non-recursive one:

\[ 1. \ \forall ys \text{ zs. } l\text{-}sorted (l\text{-}sort\text{-}naive \] ys zs)
which is hopeless as the formula is false for any unsorted list \( zs \).

oops

3.1 Step 1

type-synonym 'a l-type = 'a list × 'a list × 'a list

fun l-sort-aux :: 'a::linorder l-type ⇒ 'a l-type where
\begin{align*}
l-sort-aux (x \# xs, ys, []) &= l-sort-aux (xs, [], ys @ [x]) | \\
l-sort-aux (x \# xs, ys, z \# zs) &= \begin{cases} 
  (if x \leq z \\
  \text{then } l-sort-aux (xs, [], ys @ x \# z \# zs) \\
  \text{else } l-sort-aux (x \# xs, ys @ [z], zs)) \\
\end{cases} | \\
l-sort-aux ([], ys, zs) &= ([], ys, zs)
\end{align*}

Observe that the Cartesian product of the input types has been implemented as a product type.

3.2 Step 2

definition l-sort-in :: 'a list ⇒ 'a l-type where
\[ l-sort-in \; xs \equiv (xs, [], []) \]
definition l-sort-out :: 'a l-type ⇒ 'a list where
\[ l-sort-out \; X \equiv \text{snd} (\text{snd} \; X) \]
definition l-sort :: 'a::linorder list ⇒ 'a list where
\[ l-sort \; xs \equiv l-sort-out (l-sort-aux (l-sort-in \; xs)) \]

Since the significant inputs of function \( l-sort-naive \) match pattern \(-, [], []\), those of function \( l-sort-aux \) match pattern \((-, [], [])\), thus being in a one-to-one correspondence with the type of the first component.

The target correctness theorems can then be put into the following equivalent form:

\[ \text{l-sorted \; (l-sort \; xs)} \]

\[ \text{l-count \; x \; (l-sort \; xs) = l-count \; x \; xs} \]

3.3 Step 3

The conditional recursive equation in the definition of function \( l-sort-aux \) will equivalently be associated to two distinct introduction rules in the definition of the inductive set \( l-sort-set \), one for either truth value of the Boolean condition, handled as an additional assumption. The advantage is twofold:
simpler introduction rules are obtained, and case distinctions are saved as
rule induction is applied.

\begin{verbatim}
inductive-set l-sort-set :: 'a::linorder l-type ⇒ 'a l-type set
for X :: 'a l-type where
R0: X ∈ l-sort-set X |
R1: [(x ≠ xs, ys, [])] ∈ l-sort-set X ⇒ (xs, [], ys @ [x]) ∈ l-sort-set X |
R2: [(x ≠ xs, ys, z ≠ zs) ∈ l-sort-set X; x ≤ z] ⇒
  (xs, [], ys @ x ≠ z ≠ zs) ∈ l-sort-set X |
R3: [(x ≠ xs, ys, z ≠ zs) ∈ l-sort-set X; x ≤ z] ⇒
  (x ≠ xs, ys @ [z], zs) ∈ l-sort-set X

\end{verbatim}

3.4 Step 4

\begin{verbatim}
lemma l-sort-subset:
  assumes XY: Y ∈ l-sort-set X
  shows l-sort-set Y ⊆ l-sort-set X
proof (rule subsetI, erule l-sort-set.induct)
  show Y ∈ l-sort-set X using XY .
next
  fix x xs ys
  assume (x ≠ xs, ys, []) ∈ l-sort-set X
  thus (xs, [], ys @ [x]) ∈ l-sort-set X by (rule R1)
next
  fix x xs ys z zs
  assume (x ≠ xs, ys, z ≠ zs) ∈ l-sort-set X and x ≤ z
  thus (xs, [], ys @ x ≠ z ≠ zs) ∈ l-sort-set X by (rule R2)
next
  fix x xs ys z zs
  assume (x ≠ xs, ys, z ≠ zs) ∈ l-sort-set X and ¬ x ≤ z
  thus (x ≠ xs, ys @ [z], zs) ∈ l-sort-set X by (rule R3)
qed

lemma l-sort-aux-set: l-sort-aux X ∈ l-sort-set X
proof (induction rule: l-sort-aux.induct, simp-all del: l-sort-aux.simps(2))
  fix ys :: 'a list and zs
  show ([], ys, zs) ∈ l-sort-set ([], ys, zs) by (rule R0)
next
  fix x :: 'a and xs ys
  have (x ≠ xs, ys, []) ∈ l-sort-set (x ≠ xs, ys, []) by (rule R0)
  hence (xs, [], ys @ [x]) ∈ l-sort-set (x ≠ xs, ys, []) by (rule R1)
  hence l-sort-set (xs, [], ys @ [x]) ⊆ l-sort-set (x ≠ xs, ys, [])
    by (rule l-sort-subset)
  moreover assume l-sort-aux (xs, [], ys @ [x]) ∈ l-sort-set (xs, [], ys @ [x])
  ultimately show l-sort-aux (xs, [], ys @ [x]) ∈ l-sort-set (x ≠ xs, ys, [])
    by (rule subsetD)
next
  fix x :: 'a and xs ys z zs
  assume
\end{verbatim}

11
case1: \( x \leq z \implies \\
\text{l-sort-aux} (xs, [], yzs @ x \# z \# zs) \in \text{l-sort-set} (xs, [], yzs @ x \# z \# zs) \\
\text{and} \\
\text{case2:} \neg x \leq z \implies \\
\text{l-sort-aux} (x \# xs, yzs @ [z], zs) \in \text{l-sort-set} (x \# xs, yzs @ [z], zs) \\
\text{have} 0: (x \# xs, yzs, z \# zs) \in \text{l-sort-set} (x \# xs, yzs, z \# zs) \text{ by (rule R0)} \\
\text{show} \text{l-sort-aux} (x \# xs, yzs, z \# zs) \in \text{l-sort-set} (x \# xs, yzs, z \# zs) \\
\text{proof} \text{(cases } x \leq z, \text{ simp-all)} \\
\text{assume} x \leq z \\
\text{with } \emptyset \text{ have} (xs, [], yzs @ x \# z \# zs) \in \text{l-sort-set} (x \# xs, yzs, z \# zs) \\
\text{by (rule R2)} \\
\text{hence} \text{l-sort-set} (xs, [], yzs @ x \# z \# zs) \subseteq \text{l-sort-set} (x \# xs, yzs, z \# zs) \\
\text{by (rule l-sort-subset)} \\
\text{moreover have} \text{l-sort-aux} (xs, [], yzs @ x \# z \# zs) \in \text{l-sort-set} (xs, [], yzs @ x \# z \# zs) \text{ using case1 and } (x \leq z) \text{ by simp} \\
\text{ultimately show} \text{l-sort-aux} (xs, [], yzs @ x \# z \# zs) \in \text{l-sort-set} (x \# xs, yzs, z \# zs) \text{ by (rule subsetD)} \\
\text{next} \\
\text{assume} \neg x \leq z \\
\text{with } \emptyset \text{ have} (x \# xs, yzs @ [z], zs) \in \text{l-sort-set} (x \# xs, yzs, z \# zs) \\
\text{by (rule R3)} \\
\text{hence} \text{l-sort-set} (x \# xs, yzs @ [z], zs) \subseteq \text{l-sort-set} (x \# xs, yzs, z \# zs) \\
\text{by (rule l-sort-subset)} \\
\text{moreover have} \text{l-sort-aux} (x \# xs, yzs @ [z], zs) \in \text{l-sort-set} (x \# xs, yzs @ [z], zs) \text{ using case2 and } (\neg x \leq z) \text{ by simp} \\
\text{ultimately show} \text{l-sort-aux} (x \# xs, yzs @ [z], zs) \in \text{l-sort-set} (x \# xs, yzs, z \# zs) \text{ by (rule subsetD)} \\
\text{qed} \\
\text{qed} \\

The reader will have observed that the simplification rule arising from the second equation in the definition of function \text{l-sort-aux}, i.e. the one whose right-hand side contains a conditional, has been ignored in the initial backward steps of the previous proof. The reason is that it would actually make more complex the conclusion of the corresponding subgoal, as can easily be verified by trying to leave it enabled.

\text{lemma} \text{l-sort-aux} X \in \text{l-sort-set} X \\
\text{proof} \text{(induction rule: l-sort-aux.induct, simp-all)}

As a result of the application of the rule, the related subgoal takes the following form:

\[ \forall x xs ys z zs. \quad \| x \leq z \implies \]
\[ l\text{-}sort\text{-}aux\ (xs, [], ys @ x # z # zs) \]
\[ \in l\text{-}sort\text{-}set\ (xs, [], ys @ x # z # zs); \]
\[ \neg x \leq z \implies \]
\[ l\text{-}sort\text{-}aux\ (x # xs, ys @ [z], zs) \]
\[ \in l\text{-}sort\text{-}set\ (x # xs, ys @ [z], zs) \]
\[ \implies (x \leq z \implies \]
\[ l\text{-}sort\text{-}aux\ (xs, [], ys @ x # z # zs) \]
\[ \in l\text{-}sort\text{-}set\ (x # xs, ys @ [z], zs) \]
\[ (\neg x \leq z \implies \]
\[ l\text{-}sort\text{-}aux\ (x # xs, ys @ [z], zs) \]
\[ \in l\text{-}sort\text{-}set\ (x # xs, ys @ [z], zs) \]

A total of 3 subgoals...

Now the conclusion is comprised of a conjunction of two implications. This is pointless, since case distinction is faster than the application of conjunction and implication introduction rules in providing sufficient assumptions for the simplification of both the induction hypotheses and the conclusion.

oops

These considerations clearly do not depend on the particular function under scrutiny, so that postponing the application of conditional simplification rules to case distinction turns out to be a generally advisable strategy for the accomplishment of step 4.

3.5 Step 5

Two invariants are defined here below, one for each of the target correctness theorems:

fun \textit{l-sort-inv-1} :: "a::linorder l-type ⇒ bool where
\textit{l-sort-inv-1} x xs y z =
\ (\textit{l-sorted} y ys ∧ \textit{l-sorted} z zs) ∧
\ (\textit{last} y ys \leq x ∧ \textit{last} y ys \leq z) \ |
\ l\text{-}sort\text{-}inv-1 (x # xs, y , z, zs) =
\ \ (\textit{l-sorted} y ys ∧ \textit{last} y ys \leq x) \ |
\ l\text{-}sort\text{-}inv-1 ( , , zs) =
\ \ l\text{-}sorted zs

definition \textit{l-sort-inv-2} :: "a ⇒ "a list ⇒ "a l-type ⇒ bool where
\textit{l-sort-inv-2} x xs X ≡ (\textit{fst} X = [] \implies \textit{fst} (\textit{snd} X) = [] ) ∧
\ l\text{-}count x (\textit{fst} X) + l\text{-}count x (\textit{fst} (\textit{snd} X)) + l\text{-}count x (\textit{snd} (\textit{snd} X)) =
\ l\text{-}count x xs

13
More precisely, the second invariant, whose type has to match ‘a l-type ⇒ bool according to the method specification, shall be comprised of function l-sort-inv-2 x xs, where $x$, $xs$ are the free variables appearing in the latter target theorem.

Both of the above definitions are non-recursive; command fun is used in the former for the sole purpose of taking advantage of pattern matching.

3.6 Step 6

**lemma** l-sort-input-1: l-sort-inv-1 (xs, [], [])
*by* simp

**lemma** l-sort-input-2: l-sort-inv-2 x xs (xs, [], [])
*by* (simp add: l-sort-inv-2-def l-count-def)

3.7 Step 7

**definition** l-sort-form :: ‘a l-type ⇒ bool where
l-sort-form X ≡ fst X = []

**lemma** l-sort-intro-1:
l-sort-inv-1 X ⇒ l-sorted (l-sort-out X)
*by* (rule l-sort-inv-1.cases [of X], simp-all add: l-sort-out-def)

**lemma** l-sort-intro-2:
[l-sort-inv-2 x xs X; l-sort-form X] ⇒
l-count x (l-sort-out X) = l-count x xs

3.8 Step 8

**lemma** l-sort-form-aux-all: l-sort-form (l-sort-aux X)
*by* (rule l-sort-aux.induct [of AX. l-sort-form (l-sort-aux X)], simp-all add: l-sort-form-def)

**lemma** l-sort-form-aux: l-sort-form (l-sort-aux (xs, [], []))
*by* (rule l-sort-form-aux-all)

3.9 Step 9

The proof of the first invariance property requires the following lemma, stating that in case two lists are sorted, their concatenation still is such as long as the last item of the former is not greater than the first one of the latter.

**lemma** l-sorted-app [rule-format]:
l-sorted xs → l-sorted ys → last xs ≤ hd ys → l-sorted (xs @ ys)
proof (induction xs rule: l-sorted.induct, simp-all, (rule impI)+)
  fix x
  assume l-sorted ys and x ≤ hd ys
  thus l-sorted (x # ys) by (cases ys, simp)
qed

lemma l-sort-invariance-1:
  assumes XY: Y ∈ l-sort-set X and X: l-sort-inv-1 X
  shows l-sort-inv-1 Y
using XY
proof (rule l-sort-set.induct, simp-all)
  show l-sort-inv-1 X using X .
next
  fix x :: 'a and xs ys
  assume I: l-sort-inv-1 (x # xs, ys, [])
  show l-sorted (ys @ [x])
    proof (cases ys, simp)
      fix a as
      assume ys = a # as
      hence l-sorted ys ∧ last ys ≤ x using I by simp
      moreover have l-sorted [x] by simp
      ultimately show ?thesis by (simp add: l-sorted-app)
    qed
next
  fix x :: 'a and xs ys z zs
  assume XZ: x ≤ z and I: l-sort-inv-1 (x # xs, ys, z # zs)
  thus l-sorted (ys @ x # z # zs)
  proof (cases ys, simp)
    fix a as
    assume ys = a # as
    hence l-sorted ys ∧ l-sorted (z # zs) ∧ last ys ≤ x using I by simp
    moreover with XZ have l-sorted (x # z # zs) by simp
    ultimately show ?thesis by (simp add: l-sorted-app)
  qed
next
  fix x :: 'a and xs ys z zs
  assume ¬ x ≤ z
  hence XZ: z ≤ x by simp
  assume l-sort-inv-1 (x # xs, ys, z # zs)
  thus l-sort-inv-1 (x # xs, ys @ [z], zs)
  proof (cases ys, simp)
    assume I: l-sorted (z # zs)
    show l-sort-inv-1 (x # xs, [z], zs)
      proof (cases zs, simp)
        show z ≤ x using XZ .
      next
        fix a as
        assume zs = a # as
        moreover from this have z ≤ a ∧ l-sorted zs using I by simp
moreover have l-sorted \([z]\) by simp
ultimately show ?thesis using XZ by simp
qed
next
fix \(a\) as
assume YS: \(ys = a \# as\) and l-sort-inv-1 \((x \# xs, ys, z \# zs)\)
hence I: l-sorted ys \& l-sorted \((z \# zs)\) \& last ys \leq z by simp
have l-sorted \([z]\) by simp
hence I': l-sorted \((ys @ [z])\) using I by (simp add: l-sorted-app)
show ?thesis
proof (cases zs, simp)
  show l-sort-inv-1 \((x \# xs, ys @ [z], [])\) using I' and XZ and YS by simp
next
fix \(b\) bs
assume zs = b \# bs
moreover from this have \(z \leq b\) \& l-sorted zs using I by simp
ultimately show ?thesis using I and I' and XZ and YS by simp
qed
qed
qed

Likewise, the proof of the second invariance property calls for the following lemmas, stating that the number of occurrences of a value in a list is additive with respect to both item prepending and list concatenation.

\textbf{lemma} l-count-cons: l-count \(x\) \((y \# ys)\) = l-count \(x\) \([y]\) + l-count \(x\) ys
by (simp add: l-count-def)

\textbf{lemma} l-count-app: l-count \(x\) \((ys @ zs)\) = l-count \(x\) ys + l-count \(x\) zs
by (simp add: l-count-def)

\textbf{lemma} l-sort-invariance-2: 
  assumes XY: \(Y \in l-sort-set X\) and X: l-sort-inv-2 \(w ws X\)
  shows l-sort-inv-2 \(w ws Y\)
  using XY
proof (rule l-sort-set.induct)
  show l-sort-inv-2 \(w ws X\) using X .
next
fix \(x\) \(xs\) \(ys\)
assume l-sort-inv-2 \(w ws\) \((x \# xs, ys, [])\)
thus l-sort-inv-2 \(w ws\) \((xs, [], ys @ [x])\)
proof (simp add: l-sort-inv-2-def, subst (asm) l-count-cons, subst l-count-app)
qed (simp add: l-count-def ac-simps)
next
fix \(x\) \(xs\) \(ys\) \(z\) \(zs\)
assume l-sort-inv-2 \(w ws\) \((x \# xs, ys, z \# zs)\)
thus l-sort-inv-2 \(w ws\) \((xs, [], ys @ x \# z \# zs)\)
proof (simp add: l-sort-inv-2-def, subst (asm) l-count-cons, subst l-count-app,
3.10 Step 10

Theorem L-sorted (L-sort xs)
Proof

3.10 Step 10

Theorem L-count x (L-sort xs) = L-count x xs
Proof

4 Case study 2

Theory CaseStudy2
Imports Main ~~/src/HOL/Library/Multiset
Begin

In the second case study, the problem will be examined of defining a function l-ins performing item insertion into binary search trees (admitting value repetitions) of elements of a linear order, and then proving the cor-
rectness of this definition, i.e. that the trees output by the function still be sorted if the input ones are and contain one more occurrence of the inserted value, the number of occurrences of any other value being left unaltered.

Here below is a naive tail-recursive definition of such function:

```
datatype 'a bintree = Leaf | Branch 'a 'a bintree 'a bintree

function (sequential) t-ins-naive ::
  bool ⇒ 'a::linorder ⇒ 'a bintree list ⇒ 'a bintree

where
  t-ins-naive False x (Branch y yl yr # ts) = (if x ≤ y
    then t-ins-naive False x (yl # Branch y yl yr # ts)
    else t-ins-naive False x (yr # Branch y yl yr # ts)) |
  t-ins-naive False x (Leaf # ts) =
  t-ins-naive True x (Branch x Leaf Leaf # ts) |
  t-ins-naive True x (xt # Branch y yl yr # ts) = (if x ≤ y
    then t-ins-naive True x (Branch y xt yr # ts)
    else t-ins-naive True x (Branch y yl xt # ts)) |
  t-ins-naive True x [xt] = xt
  by pat-completeness auto
```

The list appearing as the third argument, deputed to initially contain the sole tree into which the second argument has to be inserted, is used to unfold all the involved subtrees until a leaf is reached; then, such leaf is replaced with a branch whose root value matches the second argument, and the subtree list is folded again. The information on whether unfolding or folding is taking place is conveyed by the first argument, whose value will respectively be False or True.

According to this plan, the computation is meant to terminate in correspondence with pattern True, - [-]. Hence, the above naive definition comprises a non-recursive equation for this pattern only, so that the residual ones True, -, - # Leaf # - and -, -, [] are not covered by any equation.

That which decreases in recursive calls is the size of the head of the subtree list during unfolding, and the length of the list during folding. Furthermore, unfolding precedes folding in the recursive call pipeline, viz. there is a recursive equation switching from unfolding to folding, but no one carrying out the opposite transition. These considerations suggest that a measure function suitable to prove the termination of function t-ins-naive should roughly match the sum of the length of the list and the size of the list head during unfolding, and the length of the list alone during folding.

This idea can be refined by observing that the length of the list increases by one at each recursive call during unfolding, and does not change in the recursive call leading from unfolding to folding, at which the size of the input list head (a leaf) equals zero. Therefore, in order that the measure function value be strictly decreasing in each recursive call, the size of the list head
has to be counted more than once during unfolding – e.g. twice –, and the
length of the list has to be decremented by one during folding – no more
than that, as otherwise the function value would not change in the passage
from a two-item to a one-item list.

As a result, a suitable measure function and the corresponding termina-
tion proof are as follows:

```gml
fun t-ins-naive-measure :: bool × 'a × 'a bintree list ⇒ nat where
t-ins-naive-measure (b, x, ts) = (if b
  then length ts - 1
  else length ts + 2 * size (hd ts))
termination t-ins-naive
by (relation measure t-ins-naive-measure, simp-all)
```

Some further functions are needed to express the aforesaid correctness
properties of function t-ins-naive:

```gml
primrec t-set :: 'a bintree ⇒ 'a set where
t-set Leaf = {} |
t-set (Branch x xl xr) = {x} ∪ t-set xl ∪ t-set xr
primrec t-multiset :: 'a bintree ⇒ 'a multiset where
t-multiset Leaf = {#} |
t-multiset (Branch x xl xr) = {#x#} + t-multiset xl + t-multiset xr
lemma t-set-multiset: t-set xt = set-of (t-multiset xt)
by (induction, simp-all)
primrec t-sorted :: 'a::linorder bintree ⇒ bool where
t-sorted Leaf = True |
t-sorted (Branch x xl xr) =
  ((∀ y ∈ t-set xl. y ≤ x) ∧ (∀ y ∈ t-set xr. x < y) ∧ t-sorted xl ∧ t-sorted xr)
definition t-count :: 'a ⇒ 'a bintree ⇒ nat where
t-count x xt ≡ count (t-multiset xt) x
```

Functions t-set and t-multiset return the set and the multiset, respec-
tively, of the items of the input tree; the connection between them expressed
by lemma t-set-multiset will be used in step 9.

The target correctness theorems can then be enunciated as follows:

```gml
t-sorted xt −→ t-sorted (t-ins-naive False x [xt])
t-count y (t-ins-naive False x [xt]) =
```

19
(if y = x then Suc else id) (t-count y xt)

4.1 Step 1

This time, the Cartesian product of the input types will be implemented as a record type. The second command instructs the system to regard such type as a datatype, thus enabling record patterns:

record 'a t-type =
  folding :: bool
  item :: 'a
  subtrees :: 'a bintree list

function (sequential) t-ins-aux :: 'a::linorder t-type ⇒ 'a t-type where
t-ins-aux (folding = False, item = x, subtrees = Branch y yl yr # ts) =
  (if x ≤ y
   then t-ins-aux (folding = False, item = x, subtrees = yl # Branch yyl yr # ts)
     else t-ins-aux (folding = False, item = x, subtrees = yr # Branch yyl yr # ts))
  t-ins-aux (folding = False, item = x, subtrees = Leaf # ts)
  t-ins-aux (folding = True, item = x, subtrees = x # Branch yyl yr # ts)
  (if x ≤ y
   then t-ins-aux (folding = True, item = x, subtrees = Branch yxt y # ts)
     else t-ins-aux (folding = True, item = x, subtrees = Branch yyl xt # ts))
  t-ins-aux X = X
by pat-completeness auto

Observe that the pattern appearing in the non-recursive equation matches any one of the residual patterns (folding = True, item = -, subtrees = [-]), (folding = True, item = -, subtrees = - # Leaf # -), (folding = -, item = -). subtrees = []), thus complying with the requirement that the definition of function t-ins-aux be total.

Since the arguments of recursive calls in the definition of function t-ins-aux are the same as those of function t-ins-naive, the termination proof developed for the latter can be applied to the former as well by just turning the input product type of the previous measure function into the input record type of function t-ins-aux.

fun t-ins-aux-measure :: 'a t-type ⇒ nat where
t-ins-aux-measure (folding = b, item = x, subtrees = ts) = (if b
  then length ts - 1
  else length ts + 2 * size (hd ts))

termination t-ins-aux
by \((\text{relation measure } t\text{-ins-aux-measure, simp-all})\)

### 4.2 Step 2

**definition** \(t\text{-ins-in} :: \text{'a bintree} \Rightarrow \text{'a t-type where}\)
\[
t\text{-ins-in } x \ xt \equiv (\lfolding = \text{False}, \text{item} = x, \text{subtrees} = [xt])\]

**definition** \(t\text{-ins-out} :: \text{'a t-type} \Rightarrow \text{'a bintree where}\)
\[
t\text{-ins-out } X \equiv \text{hd } (\text{subtrees } X)\]

**definition** \(t\text{-ins} :: \text{'a::linorder} \Rightarrow \text{'a bintree} \Rightarrow \text{'a bintree where}\)
\[
t\text{-ins } x \ xt \equiv t\text{-ins-out } (t\text{-ins-aux } (t\text{-ins-in } x \ xt))\]

Since the significant inputs of function \(t\text{-ins-naive}\) match pattern \(\text{False}, [-], [\cdot]\), those of function \(t\text{-ins-aux}\) match pattern \((\lfolding = \text{False}, \text{item} = -, \text{subtrees} = [\cdot])\), thus being in a one-to-one correspondence with the Cartesian product of the types of the second and the third component.

Then, the target correctness theorems can be put into the following equivalent form:

\[
t\text{-sorted } xt \rightarrow t\text{-sorted } (t\text{-ins } x \ xt)\]

\[
t\text{-count } y \ (t\text{-ins } x \ xt) = (\text{if } y = x \text{ then } \text{Suc else } \text{id}) \ (t\text{-count } y \ xt)\]

### 4.3 Step 3

**inductive-set** \(t\text{-ins-set} :: \text{'a::linorder t-type} \Rightarrow \text{'a t-type set}\)
for \(X :: \text{'a t-type where}\)
\[
R0: X \in t\text{-ins-set } X\]
\[
R1: (\lfolding = \text{False}, \text{item} = x, \text{subtrees} = \text{Branch } y \ yl \ yr \ # ts) \in t\text{-ins-set } X;\]
\[
x \leq y \implies (\lfolding = \text{False}, \text{item} = x, \text{subtrees} = \text{yl }\# \text{Branch } y \ yl \ yr \ # ts)\]
\[
\in t\text{-ins-set } X\]
\[
R2: (\lfolding = \text{False}, \text{item} = x, \text{subtrees} = \text{Branch } y \ yl \ yr \ # ts) \in t\text{-ins-set } X;\]
\[
\neg x \leq y \implies (\lfolding = \text{False}, \text{item} = x, \text{subtrees} = \text{yr }\# \text{Branch } y \ yl \ yr \ # ts)\]
\[
\in t\text{-ins-set } X\]
\[
R3: (\lfolding = \text{False}, \text{item} = x, \text{subtrees} = \text{Leaf } \# ts) \in t\text{-ins-set } X \implies\]
\[
(\lfolding = \text{True}, \text{item} = x, \text{subtrees} = \text{Branch } x \text{ Leaf } \# ts)\]
\[
\in t\text{-ins-set } X\]
\[
R4: (\lfolding = \text{True}, \text{item} = x, \text{subtrees} = xt \# \text{Branch } y \ yl \ yr \ # ts) \in t\text{-ins-set } X;\]
\[
x \leq y \implies (\lfolding = \text{True}, \text{item} = x, \text{subtrees} = \text{Branch } y \ xt \ yr \ # ts)\]
\[
\in t\text{-ins-set } X\]
\[
R5: (\lfolding = \text{True}, \text{item} = x, \text{subtrees} = xt \# \text{Branch } y \ yl \ xt \ # ts) \in t\text{-ins-set } X;\]
\[
\neg x \leq y \implies (\lfolding = \text{True}, \text{item} = x, \text{subtrees} = \text{Branch } y \ yl \ xt \ # ts)\]
\[
\in t\text{-ins-set } X\]
4.4 Step 4

lemma t-ins-subset:
  assumes XY: Y ∈ t-ins-set X
  shows t-ins-set Y ⊆ t-ins-set X
proof (rule subsetI, erule t-ins-set.induct)
  show Y ∈ t-ins-set X using XY .
next
  fix x y yl yr ts
  assume
    (folding = False, item = x, subtrees = Branch y yl yr # ts) ∈ t-ins-set X
    and x ≤ y
  thus (folding = False, item = x, subtrees = yl # Branch y yl yr # ts)
        ∈ t-ins-set X by (rule R1)
next
  fix x y yl yr ts
  assume
    (folding = False, item = x, subtrees = Leaf # ts) ∈ t-ins-set X
    and ¬ x ≤ y
  thus (folding = False, item = x, subtrees = yr # Branch y yl yr # ts)
        ∈ t-ins-set X by (rule R2)
next
  fix x ts
  assume (folding = False, item = x, subtrees = Leaf # ts) ∈ t-ins-set X
  thus (folding = True, item = x, subtrees = Branch x Leaf Leaf # ts)
        ∈ t-ins-set X by (rule R3)
next
  fix x xt y yl yr ts
  assume
    (folding = True, item = x, subtrees = xt # Branch y yl yr # ts) ∈ t-ins-set X
    and x ≤ y
  thus (folding = True, item = x, subtrees = Branch y xt yr # ts)
        ∈ t-ins-set X by (rule R4)
next
  fix x xt y yl yr ts
  assume
    (folding = True, item = x, subtrees = xt # Branch y yl yr # ts) ∈ t-ins-set X
    and ¬ x ≤ y
  thus (folding = True, item = x, subtrees = Branch y yl xt # ts)
        ∈ t-ins-set X by (rule R5)
qed

lemma t-ins-aux-set: t-ins-aux X ∈ t-ins-set X
proof (induction rule: t-ins-aux.induct,
          simp-all add: R0 del: t-ins-aux.simps(1, 3))
  fix x :: 'a and y yl yr ts
  let
    ?X = (folding = False, item = x, subtrees = Branch y yl yr # ts) and
    ?X' = (folding = False, item = x, subtrees = yl # Branch y yl yr # ts) and
    ?X'' = (folding = False, item = x, subtrees = yr # Branch y yl yr # ts)
22
assume

case1: \( x \leq y \implies t\text{-ins-aux} \ ?X' \in t\text{-ins-set} \ ?X' \) and

case2: \( \neg x \leq y \implies t\text{-ins-aux} \ ?X'' \in t\text{-ins-set} \ ?X'' \)

have 0: \( ?X \in t\text{-ins-set} \ ?X \) by (rule R0)

show t-ins-aux \( ?X \in t\text{-ins-set} \ ?X \)

proof (cases \( x \leq y \), simp-all)

assume \( x \leq y \)

with 0 have \( ?X' \in t\text{-ins-set} \ ?X \) by (rule R1)

hence \( t\text{-ins-set} \ ?X' \subseteq t\text{-ins-set} \ ?X \) by (rule t-ins-subset)

moreover have t-ins-aux \( ?X' \in t\text{-ins-set} \ ?X' \)

using case1 and \( x \leq y \) by simp

ultimately show t-ins-aux \( ?X' \in t\text{-ins-set} \ ?X \) by (rule subsetD)

next

assume \( \neg x \leq y \)

with 0 have \( ?X'' \in t\text{-ins-set} \ ?X \) by (rule R2)

hence \( t\text{-ins-set} \ ?X'' \subseteq t\text{-ins-set} \ ?X \) by (rule t-ins-subset)

moreover have t-ins-aux \( ?X'' \in t\text{-ins-set} \ ?X'' \)

using case2 and \( \neg x \leq y \) by simp

ultimately show t-ins-aux \( ?X'' \in t\text{-ins-set} \ ?X \) by (rule subsetD)

qed

next

fix \( x :: 'a \) and \( ts \)

let

\[ ?X = \{ \text{folding} = \text{False}, \text{item} = x, \text{subtrees} = \text{Leaf} \# \ ts \} \] and

\[ ?X' = \{ \text{folding} = \text{True}, \text{item} = x, \text{subtrees} = \text{Branch} \ x \ \text{Leaf} \ \text{Leaf} \ \# \ ts \} \]

have \( ?X \in t\text{-ins-set} \ ?X \) by (rule R0)

hence \( ?X' \in t\text{-ins-set} \ ?X \) by (rule R3)

hence \( t\text{-ins-set} \ ?X' \subseteq t\text{-ins-set} \ ?X \) by (rule t-ins-subset)

moreover assume t-ins-aux \( ?X' \in t\text{-ins-set} \ ?X' \)

ultimately show t-ins-aux \( ?X' \in t\text{-ins-set} \ ?X \) by (rule subsetD)

next

fix \( x :: 'a \) and \( xt \ y \ yl \ yr \ ts \)

let

\[ ?X = \{ \text{folding} = \text{True}, \text{item} = x, \text{subtrees} = \text{xt} \# \text{Branch} \ y \ \text{yl} \ \text{yr} \ \# \ ts \} \] and

\[ ?X' = \{ \text{folding} = \text{True}, \text{item} = x, \text{subtrees} = \text{Branch} \ y \ \text{xt} \ \text{yr} \ \# \ ts \} \] and

\[ ?X'' = \{ \text{folding} = \text{True}, \text{item} = x, \text{subtrees} = \text{Branch} \ yl \ xt \ \# \ ts \} \]

assume

case1: \( x \leq y \implies t\text{-ins-aux} \ ?X' \in t\text{-ins-set} \ ?X' \)

case2: \( \neg x \leq y \implies t\text{-ins-aux} \ ?X'' \in t\text{-ins-set} \ ?X'' \)

have 0: \( ?X \in t\text{-ins-set} \ ?X \) by (rule R0)

show t-ins-aux \( ?X \in t\text{-ins-set} \ ?X \)

proof (cases \( x \leq y \), simp-all)

assume \( x \leq y \)

with 0 have \( ?X' \in t\text{-ins-set} \ ?X \) by (rule R4)

hence \( t\text{-ins-set} \ ?X' \subseteq t\text{-ins-set} \ ?X \) by (rule t-ins-subset)

moreover have t-ins-aux \( ?X' \in t\text{-ins-set} \ ?X' \)

using case1 and \( x \leq y \) by simp

ultimately show t-ins-aux \( ?X' \in t\text{-ins-set} \ ?X \) by (rule subsetD)

next
assume ¬x ≤ y
with 0 have "X'' ∈ t-ins-set "X by (rule R5)
hence t-ins-set "X'' ⊆ t-ins-set "X by (rule t-ins-subset)
moreover have t-ins-aux "X'' ∈ t-ins-set "X"
using case2 and (¬ x ≤ y) by simp
ultimately show t-ins-aux "X'' ∈ t-ins-set "X by (rule subsetD)
qed

4.5 Step 5

primrec t-val :: 'a bintree ⇒ 'a where
t-val (Branch x xI xR) = x

primrec t-left :: 'a bintree ⇒ 'a bintree where
t-left (Branch x xI xR) = xI

primrec t-right :: 'a bintree ⇒ 'a bintree where
t-right (Branch x xI xR) = xR

The partiality of the definition of the previous functions, which merely return the root value and either subtree of the input branch, does not matter as they will be applied to branches only.

These functions are used to define the following invariant – this time, a single invariant for both of the target correctness theorems:

fun t-ins-inv :: 'a:linorder ⇒ 'a bintree ⇒ 'a t-type ⇒ bool where
t-ins-inv x xt (| folding = b, item = y, subtrees = ts |) =
(y = x ∧
(∀ n ∈ {...<length ts},
(t-sorted xt −→ t-sorted (ts ! n)) ∧
(0 < n −→ (∃ y yI yR. ts ! n = Branch y yI yR)) ∧
(let ts' = ts @ [Branch x xt Leaf] in t-multiset (ts ! n) =
(if b ∧ n = 0 then {#x#} else {#}) +
(if x ≤ t-val (ts' ! Suc n)
then t-multiset (t-left (ts' ! Suc n))
else t-multiset (t-right (ts' ! Suc n)))))

More precisely, the invariant, whose type has to match 'a t-type ⇒ bool according to the method specification, shall be comprised of function t-ins-inv x xt, where x, xt are the free variables appearing in the target theorems as the arguments of function t-ins.

4.6 Step 6

lemma t-ins-input: t-ins-inv x xt (| folding = False, item = x, subtrees = [xt] |)
by simp

qed
4.7 Step 7

fun t-ins-form :: 'a t-type ⇒ bool where  
t-ins-form (folding = True, item = -, subtrees = [-]) = True |  
t-ins-form (folding = True, item = -, subtrees = - # Leaf # -) = True |  
t-ins-form _. = False

lemma t-ins-intro-1:  
[t-ins-inv x xt X; t-ins-form X] ⇒  
t-sorted xt → t-sorted t-ins-out X  
proof (rule t-ins-form.cases [of X], simp-all add: t-ins-out-def)  
qed (erule conjE, drule-tac x = Suc 0 in bspec, simp-all)

lemma t-ins-intro-2:  
[t-ins-inv x xt X; t-ins-form X] ⇒  
t-count y (t-ins-out X) = (if y = x then Suc else id) (t-count y xt)  
proof (rule t-ins-form.cases [of X], simp-all add: t-ins-out-def t-count-def)  
qed (erule conjE, drule-tac x = Suc 0 in bspec, simp-all)

Defining predicate t-ins-form by means of pattern matching rather than quantifiers permits a faster proof of the introduction rules through a case distinction followed by simplification. These steps leave the subgoal corresponding to pattern (| folding = True, item = -, subtrees = - # Leaf # - |) to be proven, which can be done ad absurdum as this pattern is incompatible with the invariant, stating that all the subtrees in the list except for its head are branches.

The reason why this pattern, unlike (| folding = -, item = -, subtrees = [], |), is not filtered by predicate t-ins-form, is that the lack of its occurrences in recursive calls in correspondence with significant inputs cannot be proven by rule inversion, being it compatible with the patterns introduced by rules R3, R4, and R5.

4.8 Step 8

This step will be accomplished by first proving by recursion induction that the outputs of function t-ins-aux match either of the patterns satisfying predicate t-ins-form or else the residual one (| folding = -, item = -, subtrees = [], |), and then proving by rule inversion that the last pattern may not occur in recursive calls in correspondence with significant inputs.

definition t-ins-form-all :: 'a t-type ⇒ bool where  
t-ins-form-all X ≡ t-ins-form X ∨ subtrees X = []

lemma t-ins-form-aux-all: t-ins-form-all (t-ins-aux X)  
by (rule t-ins-aux.induct [of AX. t-ins-form-all (t-ins-aux X)],  
simp-all add: t-ins-form-all-def)
lemma t-ins-form-aux:
\[
t-ins-form (t-ins-aux [] folding = False, item = x, subtrees = [xt]))
\]
(is - (t-ins-aux ?X))
using t-ins-aux-set [of ?X]
proof (rule t-ins-set.cases, insert t-ins-form-aux-all [of ?X])
qed (simp-all add: t-ins-form-all-def)

4.9 Step 9

lemma t-ins-invariance:
\[
\begin{align*}
\text{assumes} & \quad XY: Y \in t-ins-set X \quad \text{and} \quad X: t-ins-inv x xt X \\
\text{shows} & \quad t-ins-inv x xt Y \\
\text{using} & \quad XY \\
\text{proof} (\text{rule t-ins-set}.
\end{align*}
\]
\[
\begin{align*}
\text{induct, simp-all split del: split-if})
\end{align*}
\]
\[
\begin{align*}
\text{show} & \quad t-ins-inv x xt X \\
\text{using} & \quad X. \\
\text{next} \\
\text{fix} & \quad z :: 'a::linorder \quad \text{and} \quad y yl yr ts \\
\text{assume} & \quad z = x \land \\
\forall n \in \{..<\text{Suc (length ts)}\}. \\
\text{(t-sorted xt } \rightarrow & \text{ t-sorted ((Branch y yl yr # ts) ! n)}) \land \\
\\langle 0 < n \rightarrow (\exists y' yl' yr'. ts ! (n - Suc 0) = \text{Branch y' yl' yr'}) \land \\
\text{(let ts' = Branch y yl yr # ts @ [Branch x xt Leaf] \\
\text{in t-multiset ((Branch y yl yr # ts) ! n)} = \\
\text{(if } x \leq \text{ t-val ((ts @ [Branch x xt Leaf]) ! n)} \\
\text{then t-multiset (t-left (ts' ! Suc n))} \\
\text{else t-multiset (t-right (ts' ! Suc n))))) \\
\text{(is - } \land \text{ (}\forall n \in \{..<\text{Suc (length ts)}\}. \text{?P n})) \\
\text{hence I: } \forall n \in \{..<\text{Suc (length ts)}\}. \text{?P n ..} \\
\text{assume} & \quad xy: x \leq y \\
\text{show} & \quad \forall n \in \{..<\text{Suc (Suc (length ts))}\}. \\
\text{(t-sorted xt } \rightarrow & \text{ t-sorted ((yl # Branch y yl yr # ts) ! n)}) \land \\
\\langle 0 < n \rightarrow (\exists y' yl' yr'. (Branch y yl yr # ts) ! (n - Suc 0) = \\
\text{Branch y' yl' yr'}) \land \\
\text{(let ts' = yl # Branch y yl yr # ts @ [Branch x xt Leaf] \\
\text{in t-multiset ((yl # Branch y yl yr # ts) ! n)} = \\
\text{(if } x \leq \text{ t-val ((Branch y yl yr # ts @ [Branch x xt Leaf]) ! n)} \\
\text{then t-multiset (t-left (ts' ! Suc n))} \\
\text{else t-multiset (t-right (ts' ! Suc n))))) \\
\text{(is } \forall n \in \{..<\text{Suc (Suc (length ts))}\}. \text{?Q n}) \\
\text{proof} \\
\text{fix} & \quad n \\
\text{assume} & \quad n: n \in \{..<\text{Suc (Suc (length ts))}\} \\
\text{show} & \quad ?Q n \\
\text{proof (cases n)} \\
\text{case} & \quad 0 \\
\text{have} & \quad 0 \in \{..<\text{Suc (length ts)}\} \text{ by simp} \\
\text{with I have} & \quad ?P 0 ..
thus \( ?\text{thesis} \) by (simp add: \( \text{Let-def xy 0} \))

next

\begin{itemize}
\item case (Suc \( m \))
\item hence \( m \in \{..<\text{Suc (length ts)}\} \) using \( n \) by simp
\item with \( I \) have \( ?P m .. \)
\item thus \( ?\text{thesis} \)
\end{itemize}

proof (simp add: \( \text{Let-def Suc} \))

qed (cases \( m \), simp-all)

qed

next

fix \( z :: 'a::linorder \) and \( y yl yr ts \)

assume \( z = x \land \)
\[ (\forall n \in \{..<\text{Suc (length ts)}\}. \)
\[ (\text{t-sorted } xt \rightarrow \text{t-sorted } ((\text{Branch } y yl yr \# ts) ! n)) \land \]
\[ (0 < n \rightarrow (\exists y' yl' yr'. ts! (n - \text{Suc } 0) = \text{Branch } y' yl' yr')) \land \]
\[ (\text{let } ts' = \text{Branch } y yl yr \# ts @ [\text{Branch } x xt \text{Leaf}] \)
\[ \text{in t-multiset } ((\text{Branch } y yl yr \# ts) ! n) = \]
\[ (\text{if } x \leq \text{t-val } ((ts @ [\text{Branch } x xt \text{Leaf}]) ! n) \]
\[ \text{then t-multiset } (\text{t-left } (ts' ! \text{Suc } n)) \]
\[ \text{else t-multiset } (\text{t-right } (ts' ! \text{Suc } n)))) \]
\[ (\text{is -} \land (\forall n \in \{..<\text{Suc (length ts)}\}. ?P n)) \]

hence \( I: \forall n \in \{..<\text{Suc (length ts)}\}. ?P n .. \)

assume \( xy: \neg x \leq y \)

show \( \forall n \in \{..<\text{Suc (length ts)}\}. \)
\[ (\text{t-sorted } xt \rightarrow \text{t-sorted } ((y' \# \text{Branch } yl yr \# ts) ! n)) \land \]
\[ (0 < n \rightarrow (\exists y'' yl'' yr''. (\text{Branch } y yl yr \# ts) ! (n - \text{Suc } 0) = \)
\[ \text{Branch } y'' yl'' yr'')) \land \]
\[ (\text{let } ts' = y' \# \text{Branch } y yl yr \# ts @ [\text{Branch } x xt \text{Leaf}] \)
\[ \text{in t-multiset } ((y' \# \text{Branch } y yl yr \# ts) ! n) = \]
\[ (\text{if } x \leq \text{t-val } ((\text{Branch } y yl yr \# ts @ [\text{Branch } x xt \text{Leaf}]) ! n) \]
\[ \text{then t-multiset } (\text{t-left } (ts' ! \text{Suc } n)) \]
\[ \text{else t-multiset } (\text{t-right } (ts' ! \text{Suc } n)))) \]
\[ (\text{is } \forall n \in \{..<\text{Suc (length ts)}\}. ?Q n) \]

proof

fix \( n \)

assume \( n: n \in \{..<\text{Suc (length ts)}\} \)

show \( ?Q n \)

proof (cases \( n \))

case \( 0 \)

have \( 0 \in \{..<\text{Suc (length ts)}\} \) by simp

with \( I \) have \( ?P 0 .. \)

thus \( ?\text{thesis} \) by (simp add: \( \text{Let-def xy 0} \))

next

\begin{itemize}
\item case (Suc \( m \))
\item hence \( m \in \{..<\text{Suc (length ts)}\} \) using \( n \) by simp
\item with \( I \) have \( ?P m .. \)
\item thus \( ?\text{thesis} \)
\end{itemize}
proof (simp add: Let-def Suc)
qed (cases m, simp-all)
qed
qed

next

fix z :: 'a and ts
assume z = x ∧
(∀ n ∈ {..<Suc (length ts)}).
(t-sorted xt → t-sorted ((Leaf # ts) ! n)) ∧
(0 < n → (∃ y yl yr. ts ! (n - Suc 0) = Branch y yl yr)) ∧
(let ts' = Leaf # ts @ [Branch x xt Leaf]
  in t-multiset ((Leaf # ts) ! n) =
  (if x ≤ t-val ((ts @ [Branch x xt Leaf]) ! n)
   then t-multiset (t-left (ts' ! Suc n))
   else t-multiset (t-right (ts' ! Suc n))))
(is - ∧ (∀ n ∈ {..<Suc (length ts)}. ?P n))
hence I: ∀ n ∈ {..<Suc (length ts)}. ?P n ..
show
∀ n ∈ {..<Suc (length ts)}.
(t-sorted xt → t-sorted ((Branch x Leaf Leaf # ts) ! n)) ∧
(let ts' = Branch x Leaf Leaf # ts @ [Branch x xt Leaf]
  in t-multiset ((Branch x Leaf Leaf # ts) ! n) =
  (if n = 0 then {#x#} else {#}) +
  (if x ≤ t-val ((ts @ [Branch x xt Leaf]) ! n)
   then t-multiset (t-left (ts' ! Suc n))
   else t-multiset (t-right (ts' ! Suc n))))
(is ∀ n ∈ {..<Suc (length ts)}. ?Q n)
proof
fix n
assume n: n ∈ {..<Suc (length ts)}
show ?Q n
proof (cases n)
  case 0
  have 0 ∈ {..<Suc (length ts)} by simp
  with I have ?P 0 ..
  thus ?thesis by (simp add: Let-def 0 split add: split-if-asm)
next
  case (Suc m)
  have ?P n using I and n ..
  thus ?thesis by (simp add: Let-def Suc)
qed
qed

next

fix z :: 'a and zt y yl yr ts
assume z = x ∧
(∀ n ∈ {..<Suc (Suc (length ts))}).
(t-sorted xt → t-sorted ((zt # Branch y yl yr # ts) ! n)) ∧
(0 < n → (∃ y' yl' yr'. (Branch y yl yr # ts) ! (n - Suc 0) =
  Branch y' yl' yr')) ∧
(let ts' = zt \# Branch y yl yr \# ts @ [Branch x xt Leaf] in t-multiset ((zt \# Branch y yl yr \# ts) ! n) =
\[\begin{align*}
(\text{if } n = 0 \text{ then } \{\#x\#\} \text{ else } \{\#\}) + \\
(\text{if } x \leq \text{ t-val } ((\text{Branch y yl yr} \# ts @ [\text{Branch x xt Leaf}]) ! n) \text{ then } t-multiset (t-left (ts' ! Suc n)) \\
\text{else } t-multiset (t-right (ts' ! Suc n)))
\end{align*}\]
\[(\text{is - } \forall n \in \{..,<\text{Suc (length ts)}\}. \ ?P n)\]
\text{hence I: } \forall n \in \{..,<\text{Suc (length ts)}\}. \ ?P n ..

\text{assume } xy: x \leq y

\text{show }
\forall n \in \{..,<\text{Suc (length ts)}\}.
(t-sorted xz \rightarrow t-sorted ((\text{Branch y yl yr} \# xz) ! n)) \land
\(\left(0 < n \rightarrow (\exists y' yr'. ts! (n - \text{Suc 0}) = \text{Branch y' yl' yr'}) \landight.
\left.\text{let ts' = Branch y yl yr \# ts @ [Branch x xt Leaf] in t-multiset ((\text{Branch y yl yr} \# ts) ! n) =}ight.
\(\begin{align*}
(\text{if } n = 0 \text{ then } \{\#x\#\} \text{ else } \{\#\}) + \\
(\text{if } x \leq \text{ t-val } ((\text{ts @ [Branch x xt Leaf]} ! n) \text{ then } t-multiset (t-left (ts' ! Suc n)) \\
\text{else } t-multiset (t-right (ts' ! Suc n)))
\end{align*}\)
\[(\text{is } \forall n \in \{..,<\text{Suc (length ts)}\}. \ ?P n)\]

\text{proof}
\text{fix } n

\text{assume } n: n \in \{..,<\text{Suc (length ts)}\}

\text{show } \forall Q n

\text{proof } (\text{cases } n)

\text{case } 0

\text{have } 0 \in \{..,<\text{Suc (length ts)}\} \text{ by simp}

\text{with } I \text{ have } ?P 0 ..

\text{hence I0: } (t-sorted xz \rightarrow t-sorted xz) \land
\text{t-multiset xz = } \{\#x\#\} + \text{t-multiset yl}

\text{by } (\text{simp add: Let-def xy})

\text{have } \text{Suc 0} \in \{..,<\text{Suc (length ts)}\} \text{ by simp}

\text{with } I \text{ have } ?P (\text{Suc 0}) ..

\text{hence I1: } (t-sorted xz \rightarrow t-sorted (\text{Branch y yl yr})) \land
\text{t-multiset (Branch y yl yr) =}
\(\begin{align*}
(\text{if } x \leq \text{ t-val } ((\text{ts @ [Branch x xt Leaf]} ! 0) \text{ then } t-multiset (t-left ((\text{ts @ [Branch x xt Leaf]} ! 0)) \\
\text{else } t-multiset (t-right ((\text{ts @ [Branch x xt Leaf]} ! 0)))}
\end{align*}\)

\text{by } (\text{simp add: Let-def})

\text{show } ?\text{thesis}

\text{proof } (\text{simp add: Let-def 0 del: t-sorted.simps split del: split-if, rule conjf, simp-all add: Let-def 0 del: t-sorted.simps, rule-tac [2] conjf, rule-tac [!] impI})

\text{assume } s: t-sorted xz

\text{hence t-sorted xz using I0 by simp}

\text{moreover have t-sorted (Branch y yl yr) using I1 and s by simp}

\text{moreover have t-set xz = } \{x\} \cup \text{t-set yl using I0}

\text{by } (\text{simp add: t-set-multiset})

\text{ultimately show t-sorted (Branch y yl yr) using xy by simp}
next
 assume \( x \leq t\text{-}val ((ts @ \text{Branch } x \text{ Root } y) ! 0) \)
 hence \( t\text{-}multiset (t\text{-}left ((ts @ \text{Branch } x \text{ Root } y) ! 0)) = \)
 \( t\text{-}multiset (\text{Branch } y \text{ Root } y) \) using \( \text{I1 by simp} \)
 thus \( \{\#y\#\} + t\text{-}multiset z + t\text{-}multiset yr = \)
 \( \{\#x\#\} + t\text{-}multiset (t\text{-}left ((ts @ \text{Branch } x \text{ Root } y) ! 0)) \) using \( \text{I0} \)
 by (simp add: \( \text{ac-simps} \))

next
 assume \( \neg x \leq t\text{-}val ((ts @ \text{Branch } x \text{ Root } y) ! 0) \)
 hence \( t\text{-}multiset (t\text{-}right ((ts @ \text{Branch } x \text{ Root } y) ! 0)) = \)
 \( t\text{-}multiset (\text{Branch } y \text{ Root } y) \) using \( \text{I1 by simp} \)
 thus \( \{\#y\#\} + t\text{-}multiset z + t\text{-}multiset yr = \)
 \( \{\#x\#\} + t\text{-}multiset (t\text{-}right ((ts @ \text{Branch } x \text{ Root } y) ! 0)) \) using \( \text{I0} \)
 by (simp add: \( \text{ac-simps} \))

qed

next
 case \( \text{Suc } n \)
 have \( \text{Suc } n \in \{..<\text{Suc } (\text{Suc } (\text{length } ts))\} \) using \( n \) by simp
 with \( \text{I have } ?P (\text{Suc } n) \) ..
 thus \( \text{thesis by (simp add: } \text{Let-def Suc} \)\)

qed

next
 fix \( z :: 'a \) and \( zt \text{ Root } yl \text{ Root } ts \)
 assume \( z = x \land \)
 \( (\forall n \in \{..<\text{Suc } (\text{Suc } (\text{length } ts))\} . \)
 \( (t\text{-}sorted zt \rightarrow t\text{-}sorted ((zt \# \text{Branch } y \text{ Root } yl \# ts) ! n)) \land \)
 \( (0 < n \rightarrow (\exists y' \text{ Root } yr'. (\text{Branch } y \text{ Root } yl \# ts) ! (n - \text{Suc } 0) = \)
 \( \text{Branch } y' \text{ Root } yl' \text{ Root } yr')) \land \)
 \( (\text{let ts'} = \text{Branch } y \text{ Root } zt \# ts @[\text{Branch } x \text{ Root } y] \)
 \( \text{in } t\text{-}multiset ((zt \# \text{Branch } y \text{ Root } yl \# ts) ! n) = \)
 \( (\text{if } n = 0 \text{ then } \{\#x\#\} \text{ else } \{\#\}) + \)
 \( (\text{if } x \leq t\text{-}val ((\text{Branch } y \text{ Root } yl \# ts@[\text{Branch } x \text{ Root } y]) ! n) \)
 \( \text{then } t\text{-}multiset (t\text{-}left (ts' ! \text{Suc } n)) \)
 \( \text{else } t\text{-}multiset (t\text{-}right (ts' ! \text{Suc } n)))) \) \)
 \( (\text{is } - \land (\forall n \in \{..<\text{Suc } (\text{Suc } (\text{length } ts))\}. ?P n) \)\)
 hence \( \text{I : } \forall n \in \{..<\text{Suc } (\text{Suc } (\text{length } ts))\}. ?P n \) ..
 assume \( xy: \neg x \leq y \)
 show
 \( \forall n \in \{..<\text{Suc } (\text{length } ts)\} . \)
 \( (t\text{-}sorted zt \rightarrow t\text{-}sorted ((\text{Branch } y \text{ Root } zt \# ts) ! n)) \land \)
 \( (0 < n \rightarrow (\exists y' \text{ Root } yr'. ts ! (n - \text{Suc } 0) = \text{Branch } y' \text{ Root } yl' \text{ Root } yr')) \land \)
 \( (\text{let ts'} = \text{Branch } y \text{ Root } zt \# ts @[\text{Branch } x \text{ Root } y] \)
 \( \text{in } t\text{-}multiset ((\text{Branch } y \text{ Root } zt \# ts) ! n) = \)
 \( (\text{if } n = 0 \text{ then } \{\#x\#\} \text{ else } \{\#\}) + \)
 \( (\text{if } x \leq t\text{-}val ((ts@[\text{Branch } x \text{ Root } y]) ! n) \)
 \( \text{then } t\text{-}multiset (t\text{-}left (ts' ! \text{Suc } n)) \)
 \( \text{else } t\text{-}multiset (t\text{-}right (ts' ! \text{Suc } n)))) \) \)
 \( (\text{is } \forall n \in \{..<\text{Suc } (\text{length } ts)\}. ?Q n) \)
proof
  fix n
  assume n: n ∈ {..<Suc (length ts)}
  show ?Q n
proof (cases n)
  case 0
  have 0 ∈ {..<Suc (Suc (length ts))} by simp
  with I have ?P 0 ..
  hence I0: (t-sorted xt → t-sorted zt) ∧
  t-multiset zt = {#x#} + t-multiset yr
  by (simp add: Let-def xy)
  have Suc 0 ∈ {..<Suc (Suc (length ts))} by simp
  with I have ?P (Suc 0) ..
  hence I1: (t-sorted xt → t-sorted (Branch y yl yr)) ∧
  t-multiset (Branch y yl yr) =
  (if x ≤ t-val ((ts @ [Branch x xt Leaf]) ! 0)
  then t-multiset (t-left ((ts @ [Branch x xt Leaf]) ! 0))
  else t-multiset (t-right ((ts @ [Branch x xt Leaf]) ! 0)))
  by (simp add: Let-def)
  show ?thesis
proof (simp add: Let-def 0 del: t-sorted.simps split del: split-if,
  rule conjI, simp-all add: Let-def 0 del: t-sorted.simps,
  rule-tac [2] conjI, rule-tac [] simpI)
  assume s: t-sorted xt
  hence t-sorted zt using I0 by simp
  moreover have t-sorted (Branch y yl yr) using I1 and s by simp
  moreover have t-set zt = {x} ∪ t-set yr using I0
  by (simp add: t-set-multiset)
  ultimately show t-sorted (Branch y yl zt) using xy by simp
next
  assume x ≤ t-val ((ts @ [Branch x xt Leaf]) ! 0)
  hence t-multiset (t-left ((ts @ [Branch x xt Leaf]) ! 0)) =
  t-multiset (Branch y yl yr) using I1 by simp
  thus {#y#} + t-multiset yl + t-multiset zt =
  {#x#} + t-multiset (t-left ((ts @ [Branch x xt Leaf]) ! 0)) using I0
  by (simp add: ac-simps)
next
  assume ¬ x ≤ t-val ((ts @ [Branch x xt Leaf]) ! 0)
  hence t-multiset (t-right ((ts @ [Branch x xt Leaf]) ! 0)) =
  t-multiset (Branch y yl yr) using I1 by simp
  thus {#y#} + t-multiset yl + t-multiset zt =
  {#x#} + t-multiset (t-right ((ts @ [Branch x xt Leaf]) ! 0)) using I0
  by (simp add: ac-simps)
qed
next
  case (Suc m)
  have Suc n ∈ {..<Suc (Suc (length ts))} using n by simp
  with I have ?P (Suc n) ..
  thus ?thesis by (simp add: Let-def Suc)
4.10   Step 10

\textbf{theorem} $t$-sorted $xt \rightarrow t$-sorted ($t$-ins $xt$)

\textbf{proof} –

\begin{itemize}
\item let $?X = (\{folding = False, item = x, subtrees = [xt]\})$
\item have $t$-ins-aux $?X \in t$-ins-set $?X$ \textbf{by} (rule $t$-ins-aux-set)
\item moreover have $t$-ins-inv $x$ $xt$ $?X$ \textbf{by} (rule $t$-ins-input)
\item ultimately have $t$-ins-inv $x$ $xt$ ($t$-ins-aux $?X$) \textbf{by} (rule $t$-ins-invariance)
\item moreover have $t$-ins-form ($t$-ins-aux $?X$) \textbf{by} (rule $t$-ins-form-aux)
\item ultimately have $t$-sorted $xt \rightarrow t$-sorted ($t$-ins-out ($t$-ins-aux $?X$))
\item \textbf{by} (rule $t$-ins-intro-1)
\item moreover have $?X = t$-ins-in $x$ $xt$ \textbf{by} (simp add: $t$-ins-in-def)
\item ultimately show $?\text{thesis}$ \textbf{by} (simp add: $t$-ins-def)
\end{itemize}

\textbf{qed}

\textbf{theorems} $t$-count $y$ ($t$-ins $x$ $xt$) = (if $y = x$ then Suc else id) ($t$-count $y$ $xt$)

\textbf{proof} –

\begin{itemize}
\item let $?X = (\{folding = False, item = x, subtrees = [xt]\})$
\item have $t$-ins-aux $?X \in t$-ins-set $?X$ \textbf{by} (rule $t$-ins-aux-set)
\item moreover have $t$-ins-inv $x$ $xt$ $?X$ \textbf{by} (rule $t$-ins-input)
\item ultimately have $t$-ins-inv $x$ $xt$ ($t$-ins-aux $?X$) \textbf{by} (rule $t$-ins-invariance)
\item moreover have $t$-ins-form ($t$-ins-aux $?X$) \textbf{by} (rule $t$-ins-form-aux)
\item ultimately have $t$-count $y$ ($t$-ins-out ($t$-ins-aux $?X$)) =
\item (if $y = x$ then Suc else id) ($t$-count $y$ $xt$)
\item \textbf{by} (rule $t$-ins-intro-2)
\item moreover have $?X = t$-ins-in $x$ $xt$ \textbf{by} (simp add: $t$-ins-in-def)
\item ultimately show $?\text{thesis}$ \textbf{by} (simp add: $t$-ins-def)
\end{itemize}

\textbf{qed}

end

References


