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

Pasquale Noce
Security Certification Specialist at Arjo Systems - Gep S.p.A.
pasquale dot noce dot lavoro at gmail dot com
pasquale dot noce at arjowiggins-it dot com

December 14, 2021

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

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:

- *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 *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 *f-naive* a.
- *f-naive* X_1 = *f-naive* X'_1, ..., *f-naive* X_m = *f-naive* X'_m, *f-naive* $X = Y$ be the sequence (possibly with repetitions) of the equations involved in the computation of *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-naive* a) has to be proven. If recursion induction is applied to such goal, for each $i \in \{1..m\}$, the recursive equation *f-naive* X_i = *f-naive* X'_i gives rise to subgoal P (*f-naive* X'_i) $\Longrightarrow P$ (*f-naive* X_i), trivially discharged by simplification. On the contrary, the non-recursive equation *f-naive* $X = Y$ brings about the generation of

subgoal P (*f-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-naive* X).

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

- subgoal *f-inv* $X \longrightarrow P$ (*f-naive* X) arising from equation *f-naive* $X = Y$ be tractable, and
- formula *f-inv* a can be shown to be true, so that theorem P (*f-naive* a) can be inferred from conditional *f-inv* $a \longrightarrow P$ (*f-naive* a) by *modus ponens*.

Observe that the antecedent of the conditional may not have the form *f-inv* (*f-naive* x). Otherwise, the latter requirement would ask for proving formula *f-inv* (*f-naive* a), which would be at least as hard to prove as formula P (*f-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-naive* a) would have to be solved again, which would give rise to a *regressio ad infinitum*.

The latter requirement entails that formula *f-inv* a_0 holds. Moreover, for each $i \in \{1..m\}$, in the proof of conditional *f-inv* $x \longrightarrow P$ (*f-naive* x) by recursion induction, the recursive equation *f-naive* X_i = *f-naive* X'_i brings about the generation of subgoal *f-inv* $X'_i \longrightarrow P$ (*f-naive* X'_i) $\Longrightarrow$ *f-inv* $X_i \longrightarrow P$ (*f-naive* X_i). Assuming that formula *f-inv* a_{i-1} holds, it turns out that the conclusion antecedent *f-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-naive* X_i) has to be proven.

In non-trivial cases, this requires that the assumption antecedent *f-inv* X'_i be derived from the conclusion antecedent *f-inv* X_i used as a further assumption, so that the assumption consequent P (*f-naive* X'_i) – matching P (*f-naive* X_i) by virtue of equation *f-naive* X_i = *f-naive* X'_i – can be proven by *modus ponens*. This in turn requires that *f-inv* X_i imply *f-inv* X'_i, i.e. that *f-inv* x_i imply *f-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-naive* X_i = *f-naive* X'_i, so that the supposed truth of *f-inv* a_{i-1} entails that of *f-inv* a_i.

Hence, by induction, all of formulae *f-inv a*, *f-inv* a_1, ..., *f-inv* a_m turn out to be true. On the other hand, the former requirement is verified if either the antecedent *f-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-naive X*) can be shown to be true using the antecedent as an assumption. Since formula *f-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-inv X* $\Longrightarrow$ P (*f-naive X*), or rather $[\![$*f-inv x*; *f-form x*$]\!] \Longrightarrow P$ (*f-naive x*) for a suitable predicate *f-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-form* = (λx. *True*), it admits to be put into the simpler equivalent form *f-inv x* $\Longrightarrow$ P (*f-naive x*).

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

Let:

- $'a$ be the Cartesian product of types $'a_1$, ..., $'a_n$.
- *f-set* be the inductive set of type $'a \Rightarrow 'a$ *set* defined by introduction rules *x* $\in$ *f-set x*, $X_1 \in$ *f-set x* $\Longrightarrow$ $X'_1 \in$ *f-set x*, ..., $X_m \in$ *f-set x* $\Longrightarrow$ $X'_m \in$ *f-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-inv* is at least as hard as that of proving by rule induction introduction rule $[\![y \in$ *f-set x*; *f-inv x*$]\!] \Longrightarrow$ *f-inv y* – which states that for any x such that *f-inv x* is true, *f-inv* is an invariant over inductive set *f-set x*, i.e. *f-inv y* is true for each $y \in$ *f-set x*.

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

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

Let *f-aux* be the tail-recursive function of type $'a \Rightarrow 'a$ whose definition is obtained from that of *f-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-naive* X_i = *f-naive* X'_i with *f-aux* X_i = *f-aux* X'_i for each $i \in \{1..m\}$ and non-recursive equation *f-naive* $X = Y$ with *f-aux* $X = X$.

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

- *f-in* is a function of type $'a' \Rightarrow 'a$, for a suitable type $'a'$, whose range contains all the significant inputs of function *f-naive*.
- *f-out* is a function of type $'a \Rightarrow 'b$ mapping the outputs of *f-aux* to those of *f-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-aux* and *f-out* entail that equation *f-naive* x = *f-out* (*f-aux* x) holds for any x. Particularly, *f-naive* a = *f-out* (*f-aux* a); thus, being a' an inverse image of a under *f-in*, viz. a = *f-in* a', it follows that *f-naive* $a = f\ a'$. As a result, theorem P (*f-naive* a) may be rewritten as P ($f\ a'$).

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

Therefore, *f-aux* $x \in$ *f-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-aux* x be actually available – and what is more, nothing significant could be proven on *f-aux* x for any x for which its value were undefined, and then arbitrary. The trick of making the definition of *f-aux* total by adding equations for the patterns not covered in the definition of *f-naive*, if any, guarantees that this assumption be satisfied.

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

Observe that this rule is just a refinement of a rule whose proof is required for proving conditional *f-inv* $x \longrightarrow P$ (*f-naive* x) by recursion induction, so that it does not give rise to any additional proof obligation. Moreover, it

contains neither function *f-naive* nor *f-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-aux a* is $[\![f\text{-}inv\ (f\text{-}aux\ a);\ f\text{-}form\ (f\text{-}aux\ a)]\!] \Longrightarrow P\ (f\text{-}out\ (f\text{-}aux\ a))$, which by virtue of equality $a = f\text{-}in\ a'$ and the definition of function f is equivalent to formula $[\![f\text{-}inv\ (f\text{-}aux\ a);\ f\text{-}form\ (f\text{-}aux\ a)]\!] \Longrightarrow P\ (f\ a')$. 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 $f\text{-}aux\ a \in f\text{-}set\ a$, *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 $f\text{-}aux\ a \in f\text{-}set\ a$.

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 $P\ (f\text{-}naive\ a_1\ ...\ a_n)$ 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 $f\ x = f\text{-}out\ (f\text{-}aux\ (f\text{-}in\ x))$, 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* $x \Longrightarrow X'_1 \in$ *f-set* x, ..., $X_m \in$ *f-set* $x \Longrightarrow X'_m \in$ *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-aux* $x \in$ *f-set* x; a general inference scheme, independent of the specific function *f-aux*, applies to this proof.
  First, prove lemma $y \in$ *f-set* $x \Longrightarrow$ *f-set* $y \subseteq$ *f-set* x, which can easily be done by rule induction.
  Next, applying recursion induction to goal *f-aux* $x \in$ *f-set* x and then simplifying, a subgoal $X_i \in$ *f-set* X_i arises for each non-recursive equation *f-aux* $X_i = X_i$, while a subgoal *f-aux* $X'_j \in$ *f-set* $X'_j \Longrightarrow$ *f-aux* $X'_j \in$ *f-set* X_j arises for each recursive equation *f-aux* X_j = *f-aux* X'_j.
  The former subgoals can be proven by introduction rule $x \in$ *f-set* x, the latter ones as follows: rule instantiations $X_j \in$ *f-set* X_j and $X_j \in$ *f-set* $X_j \Longrightarrow X'_j \in$ *f-set* X_j imply formula $X'_j \in$ *f-set* X_j; thus *f-set* X'_j $\subseteq$ *f-set* X_j by the aforesaid lemma; from this and subgoal assumption *f-aux* $X'_j \in$ *f-set* X'_j, subgoal conclusion *f-aux* $X'_j \in$ *f-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-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-inv a*.
  In case of failure, return to step 5 so as to suitably change the definition of predicate *f-inv*.

- *Step 7* — Prove introduction rule *f-inv* $x \Longrightarrow P$ (*f-out* x), or rather $[\![$*f-inv* x; *f-form* $x]\!] \Longrightarrow P$ (*f-out* x), where *f-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-naive* applies.
  In case of failure, return to step 5 so as to suitably change the definition of predicate *f-inv*.

- *Step 8* — In case an introduction rule of the second form has been proven in step 7, prove lemma *f-form* (*f-aux a*) by recursion induction.
  If the definition of function *f-aux* resulting from step 1 contains additional non-recursive equations whose patterns do not satisfy predicate

f-form, rule inversion can be applied to exclude that *f-aux a* may match any of such patterns, using instantiation *f-aux a* ∈ *f-set a* of the lemma proven in step 4.

- *Step 9* — Prove by rule induction introduction rule ⟦*y* ∈ *f-set x*; *f-inv x*⟧ ⟹ *f-inv y*, which states the invariance of predicate *f-inv* over inductive set *f-set x* for any *x* satisfying *f-inv*.
  In case of failure, return to step 5 so as to suitably change the definition of predicate *f-inv*.
  Observe that the order in which the proof obligations related to predicate *f-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-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 a*′) by means of the following inference scheme.
  First, derive formula *f-inv* (*f-aux a*) from introduction rule ⟦*y* ∈ *f-set x*; *f-inv x*⟧ ⟹ *f-inv y* and formulae *f-aux a* ∈ *f-set a*, *f-inv a*.
  Then, derive formula *P* (*f-out* (*f-aux a*)) from either introduction rule *f-inv x* ⟹ *P* (*f-out x*) or ⟦*f-inv x*; *f-form x*⟧ ⟹ *P* (*f-out x*) and formulae *f-inv* (*f-aux a*), *f-form* (*f-aux a*) (in the latter case).
  Finally, derive theorem *P* (*f a*′) from formulae *P* (*f-out* (*f-aux a*)), *a* = *f-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-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-sort-naive* :: *'a*::*linorder list* ⇒ *'a list* ⇒ *'a list* ⇒ *'a list* **where**
l-sort-naive (*x* # *xs*) *ys* [] = *l-sort-naive xs* [] (*ys* @ [*x*]) |
l-sort-naive (*x* # *xs*) *ys* (*z* # *zs*) = (*if x* ≤ *z*
then l-sort-naive xs [] (*ys* @ *x* # *z* # *zs*)
else l-sort-naive (*x* # *xs*) (*ys* @ [*z*]) *zs*) |
l-sort-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-sort-naive*:

fun *l-sorted* :: *'a*::*linorder list* ⇒ *bool* **where**
l-sorted (*x* # *x'* # *xs*) = (*x* ≤ *x'* ∧ *l-sorted* (*x'* # *xs*)) |
l-sorted - = *True*

definition *l-count* :: *'a* ⇒ *'a list* ⇒ *nat* **where**
l-count x xs ≡ *length* [*x'*←*xs*. *x'* = *x*]

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

l-sorted (*l-sort-naive xs* [] [])

l-count x (*l-sort-naive xs* [] []) = *l-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:

theorem *l-sorted* (*l-sort-naive xs* [] [])
⟨*proof*⟩

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**
l-sort-aux (*x* # *xs*, *ys*, []) = *l-sort-aux* (*xs*, [], *ys* @ [*x*]) |
l-sort-aux (*x* # *xs*, *ys*, *z* # *zs*) = (*if* *x* ≤ *z*
  then *l-sort-aux* (*xs*, [], *ys* @ *x* # *z* # *zs*)
  else *l-sort-aux* (*x* # *xs*, *ys* @ [*z*], *zs*)) |
l-sort-aux ([], *ys*, *zs*) = ([], *ys*, *zs*)

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* ≡ (*xs*, [], [])

definition *l-sort-out* :: *'a l-type* ⇒ *'a list* **where**
l-sort-out *X* ≡ *snd* (*snd* *X*)

definition *l-sort* :: *'a*::*linorder list* ⇒ *'a list* **where**
l-sort *xs* ≡ *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:

l-sorted (*l-sort* *xs*)

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.

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*⟧ ⟹

$(xs, [], ys \mathbin{@} x \# z \# zs) \in \textit{l-sort-set}\ X \mid$
R3: $[\![(x \# xs, ys, z \# zs) \in \textit{l-sort-set}\ X; \neg\ x \leq z]\!] \Longrightarrow$
$(x \# xs, ys \mathbin{@} [z], zs) \in \textit{l-sort-set}\ X$

3.4 Step 4

lemma *l-sort-subset*:
assumes *XY*: $Y \in \textit{l-sort-set}\ X$
shows $\textit{l-sort-set}\ Y \subseteq \textit{l-sort-set}\ X$
⟨*proof*⟩

lemma *l-sort-aux-set*: $\textit{l-sort-aux}\ X \in \textit{l-sort-set}\ X$
⟨*proof*⟩

The reader will have observed that the simplification rule arising from the second equation in the definition of function *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.

lemma $\textit{l-sort-aux}\ X \in \textit{l-sort-set}\ X$
⟨*proof*⟩

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 *l-sort-inv-1* :: *'a*::*linorder l-type* ⇒ *bool* **where**
$\textit{l-sort-inv-1}\ (x \# xs, y \# ys, z \# zs) =$
$(\textit{l-sorted}\ (y \# ys) \wedge \textit{l-sorted}\ (z \# zs) \wedge$
$\textit{last}\ (y \# ys) \leq x \wedge \textit{last}\ (y \# ys) \leq z) \mid$
$\textit{l-sort-inv-1}\ (x \# xs, y \# ys, []) =$
$(\textit{l-sorted}\ (y \# ys) \wedge \textit{last}\ (y \# ys) \leq x) \mid$
$\textit{l-sort-inv-1}\ (\text{-}, \text{-}, zs) =$
$\textit{l-sorted}\ zs$

definition *l-sort-inv-2* :: *'a* ⇒ *'a list* ⇒ *'a l-type* ⇒ *bool* **where**
$\textit{l-sort-inv-2}\ x\ xs\ X \equiv (\textit{fst}\ X = [] \longrightarrow \textit{fst}\ (\textit{snd}\ X) = []) \wedge$

l-count x (fst X) + l-count x (fst (snd X)) + l-count x (snd (snd X)) = l-count x xs

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, [], [])*
⟨*proof*⟩

lemma *l-sort-input-2*: *l-sort-inv-2 x xs (xs, [], [])*
⟨*proof*⟩

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)
⟨*proof*⟩

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
⟨*proof*⟩

3.8 Step 8

lemma *l-sort-form-aux-all*: *l-sort-form (l-sort-aux X)*
⟨*proof*⟩

lemma *l-sort-form-aux*: *l-sort-form (l-sort-aux (xs, [], []))*
⟨*proof*⟩

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 $\longrightarrow$ *l-sorted ys* $\longrightarrow$ *last xs* $\leq$ *hd ys* $\longrightarrow$ *l-sorted* (*xs* @ *ys*)
⟨*proof*⟩

lemma *l-sort-invariance-1*:
assumes *XY*: *Y* $\in$ *l-sort-set X* **and** *X*: *l-sort-inv-1 X*
shows *l-sort-inv-1 Y*
⟨*proof*⟩

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.

lemma *l-count-cons*: *l-count x* (*y* # *ys*) = *l-count x* [*y*] + *l-count x ys*
⟨*proof*⟩

lemma *l-count-app*: *l-count x* (*ys* @ *zs*) = *l-count x ys* + *l-count x zs*
⟨*proof*⟩

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*
⟨*proof*⟩

3.10 Step 10

theorem *l-sorted* (*l-sort xs*)
⟨*proof*⟩

theorem *l-count x* (*l-sort xs*) = *l-count x xs*
⟨*proof*⟩

end

4 Case study 2

theory *CaseStudy2*
imports *Main HOL−Library.Multiset*
begin

In the second case study, the problem will be examined of defining a function *t-ins* performing item insertion into binary search trees (admitting value repetitions) of elements of a linear order, and then proving the correctness 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
⟨proof⟩
```

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 termination proof are as follows:

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*
⟨*proof*⟩

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

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-mset* (*t-multiset xt*)
⟨*proof*⟩

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, respectively, 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:

t-sorted xt ⟶ *t-sorted* (*t-ins-naive False x* [*xt*])

t-count y (*t-ins-naive False x* [*xt*]) =
(*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 y yl yr # ts|)
  else t-ins-aux (|folding = False, item = x,
    subtrees = yr # Branch y yl yr # ts|)) |
t-ins-aux (|folding = False, item = x, subtrees = Leaf # ts|) =
  t-ins-aux (|folding = True, item = x, subtrees = Branch x Leaf Leaf # ts|) |
t-ins-aux (|folding = True, item = x, subtrees = xt # Branch y yl yr # ts|) =
  (if x ≤ y
  then t-ins-aux (|folding = True, item = x, subtrees = Branch y xt yr # ts|)
  else t-ins-aux (|folding = True, item = x, subtrees = Branch y yl xt # ts|)) |
t-ins-aux X = X
⟨proof⟩
```

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
⟨proof⟩
```

4.2 Step 2

definition *t-ins-in* :: *'a* ⇒ *'a bintree* ⇒ *'a t-type* **where**
t-ins-in x xt ≡ (|*folding* = *False*, *item* = *x*, *subtrees* = [*xt*]|)

definition *t-ins-out* :: *'a t-type* ⇒ *'a bintree* **where**
t-ins-out X ≡ *hd* (*subtrees X*)

definition *t-ins* :: *'a*::*linorder* ⇒ *'a bintree* ⇒ *'a bintree* **where**
t-ins x xt ≡ *t-ins-out* (*t-ins-aux* (*t-ins-in x xt*))

Since the significant inputs of function *t-ins-naive* match pattern *False*, -, [-], those of function *t-ins-aux* match pattern (|*folding* = *False*, *item* = -, *subtrees* = [-]|), 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-sorted xt ⟶ *t-sorted* (*t-ins x xt*)

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

4.3 Step 3

inductive-set *t-ins-set* :: *'a*::*linorder t-type* ⇒ *'a t-type set*
for *X* :: *'a t-type* **where**
R0: *X* ∈ *t-ins-set X* |
R1: ⟦(|*folding* = *False*, *item* = *x*, *subtrees* = *Branch y yl yr # ts*|) ∈ *t-ins-set X*;
x ≤ *y*⟧ ⟹
(|*folding* = *False*, *item* = *x*, *subtrees* = *yl # Branch y yl yr # ts*|)
∈ *t-ins-set X* |
R2: ⟦(|*folding* = *False*, *item* = *x*, *subtrees* = *Branch y yl yr # ts*|) ∈ *t-ins-set X*;
¬ *x* ≤ *y*⟧ ⟹
(|*folding* = *False*, *item* = *x*, *subtrees* = *yr # Branch y yl yr # ts*|)
∈ *t-ins-set X* |
R3: (|*folding* = *False*, *item* = *x*, *subtrees* = *Leaf # ts*|) ∈ *t-ins-set X* ⟹
(|*folding* = *True*, *item* = *x*, *subtrees* = *Branch x Leaf Leaf # ts*|)
∈ *t-ins-set X* |
R4: ⟦(|*folding* = *True*, *item* = *x*, *subtrees* = *xt # Branch y yl yr # ts*|)
∈ *t-ins-set X*; *x* ≤ *y*⟧ ⟹
(|*folding* = *True*, *item* = *x*, *subtrees* = *Branch y xt yr # ts*|) ∈ *t-ins-set X* |
R5: ⟦(|*folding* = *True*, *item* = *x*, *subtrees* = *xt # Branch y yl yr # ts*|)
∈ *t-ins-set X*; ¬ *x* ≤ *y*⟧ ⟹
(|*folding* = *True*, *item* = *x*, *subtrees* = *Branch y yl xt # ts*|) ∈ *t-ins-set X*

4.4 Step 4

lemma *t-ins-subset*:

assumes *XY*: $Y \in$ *t-ins-set X*
shows *t-ins-set Y* $\subseteq$ *t-ins-set X*
⟨*proof*⟩

lemma *t-ins-aux-set*: *t-ins-aux X* $\in$ *t-ins-set X*
⟨*proof*⟩

4.5 Step 5

primrec *t-val* :: *'a bintree* ⇒ *'a* **where**
t-val (*Branch x xl xr*) = *x*

primrec *t-left* :: *'a bintree* ⇒ *'a bintree* **where**
t-left (*Branch x xl xr*) = *xl*

primrec *t-right* :: *'a bintree* ⇒ *'a bintree* **where**
t-right (*Branch x xl 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 yl yr. ts ! n = Branch y yl 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*]|)
⟨*proof*⟩

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*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *t-ins-form-aux*:
t-ins-form (*t-ins-aux* (|*folding* = *False*, *item* = *x*, *subtrees* = [*xt*]|))

(**is** - (*t-ins-aux ?X*))
⟨*proof*⟩

4.9 Step 9

lemma *t-ins-invariance*:
  assumes *XY*: *Y* ∈ *t-ins-set X* **and** *X*: *t-ins-inv x xt X*
  shows *t-ins-inv x xt Y*
⟨*proof*⟩

4.10 Step 10

theorem *t-sorted xt* ⟶ *t-sorted* (*t-ins x xt*)
⟨*proof*⟩

theorem *t-count y* (*t-ins x xt*) = (*if y* = *x then Suc else id*) (*t-count y xt*)
⟨*proof*⟩

end

References

[1] Isabelle/hol exercises — advanced — sorting with lists and trees. http://isabelle.in.tum.de/exercises/advanced/sorting/ex.pdf.

[2] A. Krauss. *Defining Recursive Functions in Isabelle/HOL*. http://isabelle.in.tum.de/website-Isabelle2013-1/dist/Isabelle2013-1/doc/functions.pdf.

[3] T. Nipkow. *A Tutorial Introduction to Structured Isar Proofs*. http://isabelle.in.tum.de/website-Isabelle2011/dist/Isabelle2011/doc/isar-overview.pdf.

[4] T. Nipkow. *Programming and Proving in Isabelle/HOL*, Nov. 2013. http://isabelle.in.tum.de/website-Isabelle2013-1/dist/Isabelle2013-1/doc/prog-prove.pdf.

[5] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, Nov. 2013. http://isabelle.in.tum.de/website-Isabelle2013-1/dist/Isabelle2013-1/doc/tutorial.pdf.