Defining a non-concrete recursive type in HOL which includes sets.

Tanja Vos and **Doaitse Swierstra**
Utrecht University, Department of Computer science
e-mail: {tanja, doaitse}@cs.uu.nl

February 18, 2000

1 Introduction

In HOL [GM93], a set of tools is provided which – for a certain class of commonly used *concrete recursive types* – automatically carries out all the formal proofs necessary to define these types. The class of recursive types supported by these tools are:

$$\begin{array}{rcl} op & = & \mathsf{C}_1\, t_1^1 \ldots t_1^{k_1} \\ & | & \ldots \\ & | & \ldots \\ & | & \mathsf{C}_m\, t_1^m \ldots t_1^{k_m} \end{array} \tag{1.1}$$

where each t_i^j is either a type expression already defined as a type (this type must *not* include *op*), or is the name *op* itself. For recursive types where the t_i^j's do include *op*, these tools do not work, and the user has to define such a type manually. In this paper, we shall describe how we manually defined the following recursive data type to HOL:

$$\begin{array}{rcl} \texttt{Val} & = & \mathsf{NUM}\ \texttt{num} \\ & | & \mathsf{SET}\ (\texttt{Val})\texttt{set} \\ & | & \mathsf{LIST}\ (\texttt{Val})\texttt{list} \\ & | & \mathsf{TREE}\ (\texttt{Val})\texttt{ltree} \end{array}$$

Section 2 outlines the general approach one has to follow when manually defining a recursive data type in HOL as it is described in [Mel89]. Sections 2 up to 4 describe the application of this approach to the specific data type mentioned above. Section 5, subsequently, shows how functions can be defined on this data type. Section 6, finally, concludes. All the results in this technical report have been mechanically verified using the HOL theorem proving environment [GM93].

2 The general approach for defining a new type in HOL

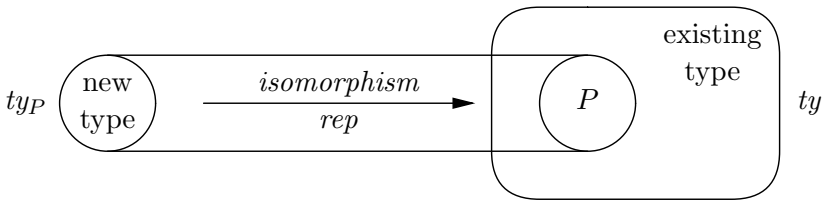
The approach to defining a new logical type as described in [Mel89], involves the following three steps:

1. find an appropriate non-empty subset of an existing type to represent the new type
2. extend the syntax of logical types to include a new type symbol, and use a type definition axiom to relate this new type to its representation
3. derive from the type definition axiom and the properties of the representing type a set of theorems that serves as an "axiomatisation" of the new type.

In the first step, a model for the new type is given by specifying a set of values that will be used to represent it. This is done by defining a predicate P on an existing type such that the set of values satisfying P is non-empty and has exactly the properties that the new type is expected to have.

In the second step, the syntax of types is extended to include a new type symbol which denotes the set of values of the new type. In HOL, this can be done by means of *type definition axioms*, a mechanism formalised by Mike Gordon in [Gor85] which defines a new type by adding a definitional axiom to the logic asserting that the new type is isomorphic to an appropriate non-empty subset of an existing type. The SML function for doing this in HOL is:

```
new_type_definition :  {name:string, pred:term, inhab_thm:thm} → thm
```



If ty is an existing type of the HOL logic, and P is a term of type $ty \rightarrow$ `bool` denoting a non-empty[1] (i.e. we can prove $\vdash \exists x :: P\ x$) subset of ty, then evaluating:

`new_type_definition :` $\{$`name = "` ty_P`"`, `pred =` P, `inhab_thm =` $\vdash \exists x :: P\ x\ \}$

results in ty_P being declared as a new type symbol characterised by the following definitional axiom:

$$\vdash \exists rep :: (\forall x\ y :: (rep.x = rep.y) \Rightarrow x = y) \\ \wedge (\forall r :: P.r = (\exists x :: r = rep.x)) \qquad (ty_P_\mathsf{TY_DEF})$$

[1] Due to the formalisation of Hilbert's ε-operator, HOL types must be non-empty (see Appendix A).

which states that the set of values denoted by the new type is isomorphic to, and consequently has the same properties as, the subset of *ty* specified by P. By adding this definition to the logic, the new type ty_P is defined in terms of existing type *ty*, and the isomorphism *rep* can be thought of as a representation function that maps a value of the new type ty_P to the value of type *ty* that represents it. The type definition axiom (ty_P_TY_DEF) above, asserts only the *existence* of a bijection from ty_P to the corresponding subset of *ty*. To introduce constants that in fact denote this isomorphism and its inverse, the following SML function is provided:

```
define_new_type_bijections :
            {ABS:string, REP:string, name:string, tyax:thm} → thm
```

If REP_ty_P and ABS_ty_P are the desired names under which to store the representation function and its inverse, then evaluating:

```
define_new_type_bijections
          {ABS = "ABS_tyP", REP = "REP_tyP",
           name = "tyP_ISO_DEF", tyax = tyP_TY_DEF}
```

defines two constants REP_ty_P:$ty \rightarrow ty_P$ and ABS_ty_P:$ty_P \rightarrow ty$, and creates the following theorem which is stored under the name ty_P_ISO_DEF:

$$\vdash (\forall a :: \mathsf{ABS}_ty_P.(\mathsf{REP}_ty_P.a) = a) \\ \wedge (\forall r :: P.r = (\mathsf{REP}_ty_P.(\mathsf{ABS}_ty_P.r) = r)) \qquad ty_P_\mathsf{ISO_DEF}$$

It is straightforward to prove that the representation and abstraction functions are injective (one-to-one) and surjective (onto), using provided SML functions:

$$\vdash (\forall a\, a' :: (\mathsf{REP}_ty_P.a = \mathsf{REP}_ty_P.a') = (a = a')) \qquad ty_P_\mathsf{REP_ONE_ONE}$$
$$\vdash \forall r :: P.r = (\exists a :: r = \mathsf{REP}_ty_P.a) \qquad ty_P_\mathsf{REP_ONTO}$$
$$\vdash \forall r\, r' :: P.r \Rightarrow P.r' \Rightarrow ((\mathsf{ABS}_ty_P.r = \mathsf{ABS}_ty_P.r') = (r = r')) \qquad ty_P_\mathsf{ABS_ONE_ONE}$$
$$\vdash \forall a :: \exists r :: (a = \mathsf{ABS}_ty_P.r) \wedge P.r \qquad ty_P_\mathsf{ABS_ONTO}$$

So, after the second step we actually have the new type ty_P, we know that the values of this type have exactly the same properties as the values in the subset P of *ty*, and we have a representation and abstraction function to go back and forth between ty_P and *ty*. Consequently, stating that some property H is true for all elements of the new type ty_P is equal to stating that for all elements in P, H is true of their image under ABS_ty_P:

$$\vdash (\forall x :: (H.x)) = (\forall r :: (P.r) \Rightarrow (H.(\mathsf{ABS}_ty_P.r))) \qquad ty_P_\mathsf{PROP}$$

In the third step, a collection of theorems is proved that state abstract characterisations of the new type. These characterisations capture the essential properties of the new type without reference to the way its values are represented and therefore acts as an abstract "axiomatisation" of it. For an inductive type σ, the assertion of the unique existence of a function g satisfying a recursion equation whose form coincides with the primitive recursion scheme of this type σ – that is, g is a paramorphism [Mee90] – provides an adequate and complete abstract characterisation for σ. From this characterisation it follows that every value of σ is constructed by one or more applications of σ's constructors, and consequently completely determines the values of σ up to isomorphism without reference to the way these are represented. Moreover, in [Mee90] it is proved that all functions with source type σ are expressible in the form of paramorphism g.

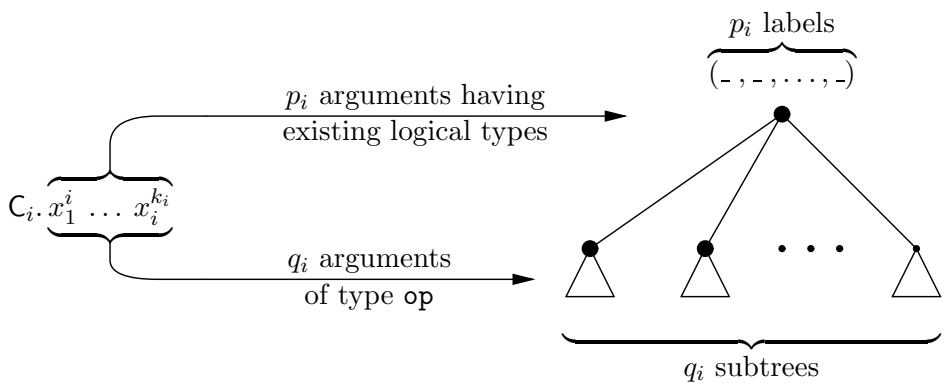
3 The representation and type definition

In [Mel89] a concrete recursive type of the form:

$$\begin{array}{rcl} \texttt{op} & = & \mathsf{C}_1\, t_1^1 \ldots t_1^{k_1} \\ & | & \ldots \\ & | & \mathsf{C}_m\, t_1^m \ldots t_1^{k_m} \end{array} \tag{3.1}$$

is represented by a non-empty subset of labelled trees (see Appendix E for labelled trees (`ltree`)). Each of the m constructors C_i, $1 \leq i \leq m$, of the concrete recursive data type is represented by a labelled tree using the scheme outlined below:

Let us consider the following instantiation of the ith constructor: $\mathsf{C}_i.x_i^1 \ldots x_i^{k_i}$, where each x_i^j is of type t_i^j which can be an existing logical type, or is the type `op` itself. Let p_i denote the number of arguments which have existing types and let q_i be the number of arguments which have type `op`, where $p_i + q_i = k_i$, then the abstract value of `op` denoted by $\mathsf{C}_i.x_i^1 \ldots x_i^{k_i}$ can be represented by a labelled tree which has p_i values associated with its root node, and q_i subtrees (for the recursive occurences of `op`). In a diagram:

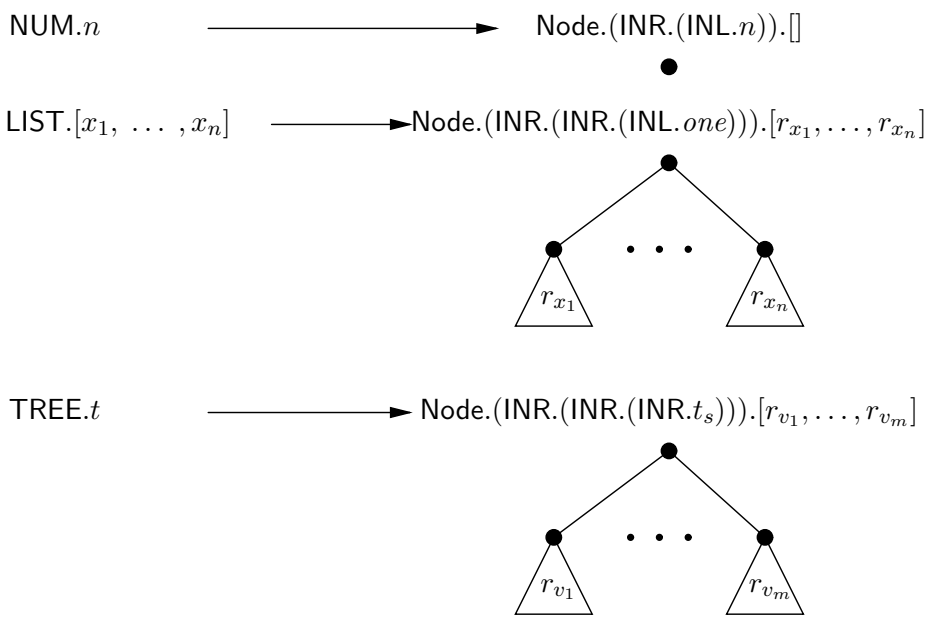


the root of the representing tree is labelled by a p_i-tuple of values. Each of these values is one of the p_i arguments to C_i which are not of type `op`. When $p_i = 0$, the representing tree is labelled with *one*, the one and only element of type `one`. The q_i subtrees shown in the diagram are the representations of the arguments of C_i that have type `op`. When $q_i = 0$, the tree will have no subtrees. Each of the m constructors can be represented by a labelled tree in this way, and consequently the representing type for `op` will be:

$$\overbrace{(\,\underbrace{(\,_\,\#\,\ldots\,\#\,_\,)}_{\text{product of } p_1 \text{ types}}\ \texttt{+}\ \ldots\ \texttt{+}\ \underbrace{(\,_\,\#\,\ldots\,\#\,_\,)}_{\text{product of } p_m \text{ types}}\,)}^{\text{sum of } m \text{ products}}\,)\texttt{ltree}$$

The subset predicate P can now be defined to specify a subset of labelled subtrees of the above type.

This method from [Mel89] only has to be adjusted a bit, in order to represent a subset of the new data type `Val`. Let us be ignorant of the sets for a while, and start using the ideas outlined above. That is, we use ((`one + num + one + tree`))`ltree` as the representing type, and make representations for the constructors NUM, LIST and TREE as follows:

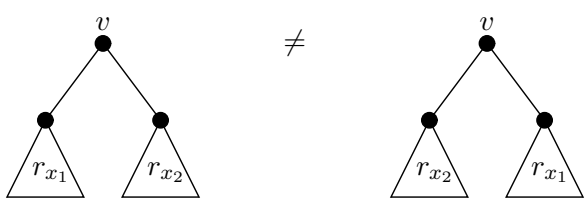


where, r_x denotes the representation as a ((`one + num + one + tree`))`ltree` of value x of type `Val`. The t_s in the root of the representation tree of TREE t denotes the shape of the tree t, and $[v_1, \ldots, v_m]$ is the list of `Val` typed values stored at the nodes of tree t. Although t_s is not an argument to the constructor TREE we can use this position at the root of the representation tree to store the shape of the `Val` tree which we obviously need in order to be able to go from the representation as a ((`one + num + one + tree`))`ltree` – where all the values of the `Val` tree are put in a list not containing any information about the shape of the original tree – to an abstract value of type `Val`.

As already indicated, the sets in the new type `Val` constitue a problem when proceeding with the method outlined above. When representing SET.$\{x_1, \ldots, x_n\}$ as a labelled ltree of which the subtrees are the representations of the values x_1,...,x_n the resulting representation function is *not* an injection, since:

$$\mathsf{SET}.\{x_1, x_2\} = \mathsf{SET}.\{x_2, x_1\}$$

but,



The solution is to represent SET.$\{x_1, \ldots, x_n\}$ by an equivalence class of ltrees in which ltrees like the two above are considered to be equivalent. Consequently, the existing type to represent our new type `Val` by shall consist of equivalence classes of ltrees, that is:

((`one + num + one + tree`))`ltree` $\rightarrow$ `bool`

Before the subset predicate can be defined, we first need to formalise the equivalence relation equiv, that given an ltree of type ((`one + num + one + tree`))`ltree`, returns the equivalence class of that ltree:

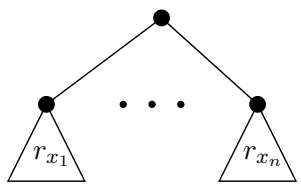
equiv : ((`one + num + one + tree`))`ltree` →
((`one + num + one + tree`))`ltree` → `bool`

The representation of a:

NUM.n value obviously has to consist of the equivalence class containing only the ltree (Node.(INR.(INL.n)).[]). Consequently, equiv.(Node.(INR.(INL.n)).[]) must return a function that only delivers *true* for argument (Node.(INR.(INL.n)).[]).

SET.$\{x_1, \ldots, x_n\}$ value, has to consist of the class containing all ltrees that are equivalent to:

Node.(INL.*one*).$[r_{x_1}, \ldots, r_{x_n}]$



For the SET case this must be the class of ltrees:

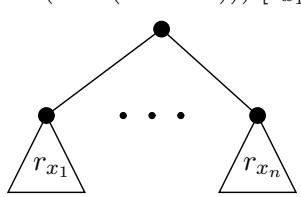
- that have (INL.*one*) at their root
- of which the **sets** of images of their subtrees under equivalence are identical. Note that, because of the absence of ordering in sets, the requirement that these particular sets are identical ensures that two ltrees as displayed on page 5 are equivalent.

Consequently, equiv.(Node.(INL.*one*).tl_1) must return a function that only delivers *true* when given an argument (Node.(INL.*one*).tl_2) such that:

image.equiv.(l2s.tl_1) = image.equiv.(l2s.tl_2)

LIST.xs value, has to consist of the class containing all ltrees that are equivalent to:

Node.(INR.(INR.(INL.*one*))).$[r_{x_1}, \ldots, r_{x_n}]$



For the LIST case this must be the class of ltrees:

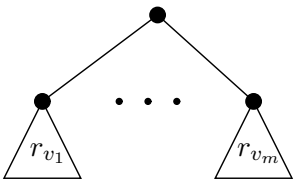
- have (INR.(INR.(INL.*one*))) at their root
- of which the **list** of images of their subtrees under equivalence are identical.

Consequently, equiv.(Node.(INR.(INR.(INL.*one*))).tl_1) must return a function that only delivers *true* for an argument (Node.(INR.(INR.(INL.*one*))).tl_2) such that:

map.equiv.tl_1 = map.equiv.tl_2

$\mathsf{TREE}.t$ value, has to consist of the class containing all ltrees that are equivalent to:

$$\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.t_s))).[r_{v_1}, \ldots, r_{v_m}]$$



Where t_s denotes the shape of tree t, and $[v_1, \ldots, v_m]$ is the list of `Val` typed values stored at the nodes of tree t. For the TREE case this must be the class of ltrees:

- have $(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.t_s)))$ at their root
- of which the **list** of images of their subtrees under equivalence are identical.

Consequently, $\mathsf{equiv}.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.t_s))).tl_1)$, must return a function that only delivers *true* for an argument $(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.t_s))).tl_2)$ such that:

$\mathsf{map.equiv}.tl_1 = \mathsf{map.equiv}.tl_2$

Below the formal definition of equiv is given:

Definition 3.1 EQUIVALENCE RELATION *equiv_DEF*

$\mathsf{equiv}.(\mathsf{Node}.v_1.tl_1).(\mathsf{Node}.v_2.tl_2) =$

$$
\begin{array}{l}
\quad (v_1 = v_2) \\
\wedge \\
\quad (\ (tl_1 = tl_2 \wedge (\exists n :: \ v_1 = \mathsf{INR}.(\mathsf{INL}.n))) \\
\qquad \vee \\
\qquad (\mathsf{image.equiv}.(\mathsf{l2s}.tl_1) = \mathsf{image.equiv}.(\mathsf{l2s}.tl_2) \wedge \mathsf{ISL}.v_1) \\
\qquad \vee \\
\qquad (\ (\mathsf{map.equiv}.tl_1 = \mathsf{map.equiv}.tl_2) \\
\qquad\quad \wedge \\
\qquad\quad (v_1 = \mathsf{INR}.(\mathsf{INR}.(\mathsf{INL}.one)) \vee \exists t :: \ v_1 = \mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.t))) \\
\qquad)\\
\quad)
\end{array}
$$

◀

Proving that the relation equiv is an equivalence relation is tedious but straightforward. Using the very nice way to represent equivalence relations from [Har93], we have:

Theorem 3.2 equiv IS AN EQUIVALENCE RELATION *equiv_EQUIV_REL*

$\mathsf{equiv}.t_1.t_2 = (\mathsf{equiv}.t_1 = \mathsf{equiv}.t_2)$

◀

The subset predicate P that has to specify a non-empty subset of equivalence classes of ltrees can now be defined as the quotient set of an appropriate subset Q of ltrees by the equivalence relation equiv. Looking at the representations of the different `Val` values, we can see that this Q must contain ltrees $(\mathsf{Node}.v.tl)$ for which hold that:

(1) if $(v = \mathsf{INR}.(\mathsf{INL}.n))$ for some number n at their root , then $tl = []$.
(2) if $(v = \mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.t)))$ for some tree t, then t and tl form an ltree
(3) for all ltrees in tl, (1) and (2) from above also hold.

in a formula:

Definition 3.3 *Q_DEF*

$$
\begin{aligned}
& Q.(\mathsf{Node}.v.tl) \;=\; \\
& \quad (\exists n :: (v = \mathsf{INR}.(\mathsf{INL}.n))) \Rightarrow tl = [] \\
& \wedge \quad (\exists t :: \; v = \mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.t))) \Rightarrow \mathsf{Is_ltree}.(\mathsf{OUTR}.(\mathsf{OUTR}.(\mathsf{OUTR}.v)), tl) \\
& \wedge \quad (\forall t :: \; t \in tl \Rightarrow Q.t)
\end{aligned}
$$

◀

Finally, the subset predicate P is defined as the quotient set of Q by equiv:

Definition 3.4 *Is_pvt_REP*

$P \;=\; Q/\mathsf{equiv}$

Theorem 3.5 *Is_pvt_REP_THM*

$P \;=\; (\lambda s.\, \exists t :: \; (s \;=\; \mathsf{equiv}.t) \wedge (Q.t))$

◀

It is not hard to prove that P is not empty, and consequently we can use the SML functions `new_type_definition` and `define_new_type_bijections` to extend the syntax of logical types to include our new type `Val`, define the type bijections ABS_`Val` and REP_`Val` between `Val` and P, and prove that these are injective and surjective:

Definition 3.6 Val_ISO_DEF

$(\forall a :: \; \mathsf{ABS_Val}.(\mathsf{REP_Val}.a) = a) \wedge (\forall r :: \; P.r = (\mathsf{REP_Val}.(\mathsf{ABS_Val}.r) = r))$

Theorem 3.7 Val_REP_ONE_ONE

$(\forall a\, a' :: \; (\mathsf{REP_Val}.a = \mathsf{REP_Val}.a') = (a = a'))$

Theorem 3.8 Val_REP_ONTO

$\forall r :: \; P.r = (\exists a :: \; r = \mathsf{REP_Val}.a)$

Theorem 3.9 Val_ABS_ONE_ONE

$\forall r\, r' :: \; P.r \Rightarrow P.r' \Rightarrow ((\mathsf{ABS_Val}.r = \mathsf{ABS_Val}.r') = (r \;=\; r'))$

Theorem 3.10 Val_ABS_ONTO

$\forall a :: \; \exists r :: \; (a \;=\; \mathsf{ABS_Val}.r) \wedge P.r$

Theorem 3.11 Val_PROP

$(\forall x :: \; (H.x)) = (\forall r :: \; (P.r) \Rightarrow (H.(\mathsf{ABS_Val}.r)))$

◀

4 The axiomatisation

The abstract axiomatisation of `Val` will be based upon four constructors:

$$\begin{array}{lcl} \mathsf{NUM} & : & \texttt{num} \to \texttt{Val} \\ \mathsf{SET} & : & (\texttt{Val})\texttt{set} \to \texttt{Val} \\ \mathsf{LIST} & : & (\texttt{Val})\texttt{list} \to \texttt{Val} \\ \mathsf{TREE} & : & (\texttt{Val})\texttt{ltree} \to \texttt{Val} \end{array}$$

To define the constructors, we need a function that given an equivalence class of ltrees returns an element of that equivalence class. We will call this function pick, and define it using Hilbert's ε-operator (see Appendix A). It satisfies the following properties:

Definition 4.1 pick *pick*

$\mathsf{pick}.c = \varepsilon t.\ c.t$

Theorem 4.2 *equiv_pick_REP_pvt*

$\mathsf{equiv} \circ \mathsf{pick} \circ \mathsf{REP_Val} = \mathsf{REP_Val}$

Theorem 4.3 *Q_pick_REP_pvt*

$\forall x :: Q.((\mathsf{pick} \circ \mathsf{REP_Val}).x)$

◀

Now the constructors can be defined as follows (see Appendix D for the definition of s2l).

Definition 4.4 *NUM_DEF*

$\mathsf{NUM}.n = \mathsf{ABS_Val}.(\mathsf{equiv}.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INL}.n)).[]))$

Definition 4.5 *SET_DEF*

$$\begin{aligned} \mathsf{SET}.s = \mathsf{ABS_Val}.(\mathsf{equiv}.(\mathsf{Node}.(\mathsf{INL}.one) \\ .(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP_Val}).(\mathsf{s2l}.s)))) \end{aligned}$$

Definition 4.6 *LIST_DEF*

$$\begin{aligned} \mathsf{LIST}.l = \mathsf{ABS_Val}.(\mathsf{equiv}.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INL}.one))) \\ .(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP_Val}).l))) \end{aligned}$$

Definition 4.7 *TREE_DEF*

$$\begin{aligned} \mathsf{TREE}.t = \mathsf{ABS_Val}.(\mathsf{equiv}.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{shape}.t))))) \\ .(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP_Val}).(\mathsf{values}.t))) \end{aligned}$$

◀

Having defined the constructors, we can state the theorem which abstractly characterises the new type `Val`. Following [Mee90] we characterise our new type by stating the unique existence of a paramorphism para as follows:

Theorem 4.8 ABSTRACT CHARACTERISATION OF `Val` *put_Axiom*

$$
\begin{array}{l}
\forall f_n\ f_s\ f_l\ f_t :: \\
\quad \exists! \mathsf{para} :: \\
\qquad (\forall n :: \mathsf{para}.(\mathsf{NUM}.n) = f_n.n) \\
\quad \wedge \\
\qquad (\forall s :: (\mathsf{FINITE}.s) \Rightarrow (\mathsf{para}.(\mathsf{SET}.s) = f_s.(\mathsf{image}.(\mathsf{split}.\mathsf{para}).s))) \\
\quad \wedge \\
\qquad (\forall l :: \mathsf{para}.(\mathsf{LIST}.l) = f_l.(\mathsf{map}.(\mathsf{split}.\mathsf{para}).l)) \\
\quad \wedge \\
\qquad (\forall t :: \mathsf{para}.(\mathsf{TREE}.t) = f_t.(\mathsf{map_tree}.(\mathsf{split}.\mathsf{para}).t))
\end{array}
$$

◀

where:

Definition 4.9 SPLIT *split*

$\forall f\ x :: \mathsf{split}.f.x \ = \ (f.x, x)$

◀

The proof of Theorem 4.8 consists of two parts, the proof of the existence of a paramorphism para, and the proof that such a paramorphism is unique.

The existence proof is based upon the following theorem about quotient sets. Informally, this theorem states that: for all equivalence relations E on α, and subsets Q on α, if ABS and REP are mutually inverse bijections between some set β and Q/equiv, and h is a function of type $\alpha \to \gamma$ that does not disthinguish between different elements in the same equivalence class, then there exists a unique function g of type $\beta \to \gamma$ such that the following diagram commutes:

$$
\begin{tikzcd}
Q \arrow[r, "E"] \arrow[d, "h"'] & Q/E \arrow[d, "\mathtt{ABS}"'] \\
\gamma & \beta \arrow[l, "g"] \arrow[u, "\mathtt{REP}"']
\end{tikzcd}
$$

Theorem 4.10 QUOTIENT SETS *QUOTIENT_THM*

For all equivalence relations E on α; for all Q defining a subset of α; for all $\mathsf{ABS} : (\alpha \to \mathtt{bool}) \to \beta$ and $\mathsf{REP} : \beta \to (\alpha \to \mathtt{bool})$, abstraction and representation functions respectively, the following theorem holds for all functions h of type $\alpha \to \gamma$:

$$
\frac{\begin{array}{c}(\forall a :: \mathsf{ABS}.(\mathsf{REP}.a) = a) \ \wedge \ (\forall r :: ((Q/E).r) = (\mathsf{REP}.(\mathsf{ABS}.r) = r)) \\ (\forall t_1\ t_2 :: (E.t_1.t_2) \ \Rightarrow \ (h.t_1 = h.t_2))\end{array}}{\exists! g :: \forall t :: (Q.t) \ \Rightarrow \ (g.(\mathsf{ABS}.(E.t)) = (h.t))}
$$

◀

Instantiating Theorem 4.10 with $((\texttt{one}+\texttt{num}+\texttt{one}+\texttt{tree}))\texttt{ltree}$ for α, equiv for E, $\mathsf{ABS_Val}$ for ABS, $\mathsf{REP_Val}$ for REP, and Q (Definition 3.3) for Q, obviously makes g a good candidate for para. Applying modus ponens to this instantiation and $\texttt{Val_ISO_DEF}$ gives us a unique function g of type $\texttt{Val} \to \gamma$ for which it holds that:

$$\frac{\forall t_1\ t_2 ::\ (\mathsf{equiv}.t_1.t_2)\ \Rightarrow\ (h.t_1 = h.t_2)}{\forall t ::\ (Q.t)\ \Rightarrow\ (g.(\mathsf{ABS_Val}.(\mathsf{equiv}.t)) = (h.t))} \tag{4.1}$$

Using theorem 4.3 (and Theorem E.3), it is easy to prove that:

Theorem 4.11 *Q_NUM_REP*
$\forall n ::\ Q\ (\mathsf{Node}.(\mathsf{INR}.(\mathsf{INL}.n)).[])$

Theorem 4.12 *Q_SET_REP*
$\forall s ::\ Q.(\mathsf{Node}.(\mathsf{INL}.one).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP_Val}).(\mathsf{s2l}.s)))$

Theorem 4.13 *Q_LIST_REP*
$\forall l ::\ Q.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INL}.one))).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP_Val}).l))$

Theorem 4.14 *Q_TREE_REP*
$\forall t ::\ Q.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{shape}.t)))).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP_Val}).(\mathsf{values}.t)))$

◀

These theorems together with (4.1) and the definitions of the constructors give us:

$$\forall t_1\ t_2 ::\ (\mathsf{equiv}.t_1.t_2)\ \Rightarrow\ (h.t_1 = h.t_2)$$

$$\begin{array}{lcl}
\forall n ::\ g.(\mathsf{NUM}.n) & = & h.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INL}.n)).[]) \\
\forall s ::\ g.(\mathsf{SET}.s) & = & h.(\mathsf{Node}.(\mathsf{INL}.one).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP_Val}).(\mathsf{s2l}.s))) \\
\forall l ::\ g.(\mathsf{LIST}.l) & = & h.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INL}.one))).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP_Val}).l)) \\
\forall t ::\ g.(\mathsf{TREE}.t) & = & h.\ (\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{shape}.t)))) \\
 & & \qquad .(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP_Val}).(\mathsf{values}.t)))
\end{array}$$

Consequently, in order to finish the existence part of the proof of Theorem 4.8, we have to find a function h, that satisfies the following properties:

(***i***). $\forall t_1\ t_2 ::\ (\mathsf{equiv}.t_1.t_2)\ \Rightarrow\ (h.t_1 = h.t_2)$
(***ii***). $h.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INL}.n)).[])$
$= f_n.n$
(***iii***). $h.(\mathsf{Node}.(\mathsf{INL}.one).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP_Val}).(\mathsf{s2l}.s)))$
$= f_s.(\mathsf{image}.(\mathsf{split}.g).s)$, for all finite sets s
(***iv***). $h.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INL}.one))).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP_Val}).l))$
$= f_l.(\mathsf{map}.(\mathsf{split.para}).l)$
(***v***). $h.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{shape}.t)))).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP_Val}).(\mathsf{values}.t)))$
$= f_t.(\mathsf{map_tree}.(\mathsf{split.para}).t)$

We claim that the following function, defined by "primitive recursion" on ltrees, satisfies these conditions, and to finish the proof of the existence part of theorem 4.8, it only remains for us to validate this claim:

$$
\begin{array}{l}
\forall v\ tl :: \ h.(\mathsf{Node}.v.tl) = k.(\mathsf{map}.h.tl).v.tl, \text{ where:} \\
\quad k = \lambda xs\ v\ tl. \\
\quad\ \mathsf{ISL}.v \rightarrow f_s.(\mathsf{l2s}.(\mathsf{zip}.(xs, (\mathsf{map}.(\mathsf{ABS_Val} \circ \mathsf{equiv}).tl)))) \\
\qquad\qquad \mid\ \mathsf{ISL}.(\mathsf{OUTR}.v) \texttt{=>} f_n.(\mathsf{OUTL}.(\mathsf{OUTR}.v)) \\
\qquad\qquad \mid\ \mathsf{ISL}.(\mathsf{OUTR}.(\mathsf{OUTR}.v)) \texttt{=>} f_l.(\mathsf{zip}.(xs, (\mathsf{map}.(\mathsf{ABS_Val} \circ \mathsf{equiv}).tl))) \\
\qquad\qquad \mid\ f_t.(\mathsf{zip_tree} \\
\qquad\qquad\qquad ((\mathsf{ABS_ltree}.(\mathsf{OUTR}.(\mathsf{OUTR}.(\mathsf{OUTR}.v)), xs)), \\
\qquad\qquad\qquad (\mathsf{ABS_ltree}.(\mathsf{OUTR}.(\mathsf{OUTR}.(\mathsf{OUTR}.v)), \mathsf{map}.(\mathsf{ABS_Val} \circ \mathsf{equiv}).tl))))
\end{array}
$$

In order to prove requirement (***i***), we have the following theorem, the proof of which is straightforward and tedious and hence will not be given here.

Theorem 4.15 *ltree_Axiom_PRESERVES_equiv*

For all functions h of type $((\texttt{one + num + one + tree}))\texttt{ltree} \rightarrow \gamma$ defined by "primitive recursion" on $((\texttt{one + num + one + tree}))\texttt{ltrees}$ (i.e. having the form $(\forall v\ tl :: \ h.(\mathsf{Node}.v.tl) = k.(\mathsf{map}.h.tl).v.tl)$ for an arbitrary function k of type $(\gamma)\texttt{list} \rightarrow ((\texttt{one + num + one + tree}))\texttt{list} \rightarrow \gamma)$:

$$
\begin{array}{l}
\forall xs_1\ xs_2\ tl_1\ tl_2\ v :: \\
\quad (\mathsf{equiv}.(\mathsf{Node}.v.tl_1).(\mathsf{Node}.v.tl_2) \wedge \\
\quad\ (\mathsf{ISL}.v) \Rightarrow (\mathsf{length}.xs_1 = \mathsf{length}.tl_1) \wedge (\mathsf{length}.xs_2 = \mathsf{length}.tl_2) \wedge \\
\qquad\qquad\qquad ((\mathsf{l2s}.(\mathsf{zip}.(xs_1, \mathsf{map.equiv}.tl_1))) = (\mathsf{l2s}.(\mathsf{zip}.(xs_2, \mathsf{map.equiv}.tl_2))) \\
\quad\ (\mathsf{ISR}.v) \Rightarrow (xs_1 = xs_2)) \\
\quad \Rightarrow \\
\quad ((k.xs_1.v.tl_1) = (k.xs_2.v.tl_2)) \\
\hline
\qquad\qquad\qquad \forall t_1\ t_2 :: (\mathsf{equiv}.t_1.t_2) \Rightarrow (h.t_1 = h.t_2)
\end{array}
$$

◀

Proving that our function k satisfies the premise of theorem 4.15 is again straightforward and tedious since, as the proof of theorem 4.15 itself, it involves lots of lemmas involving zip. We shall not give this proof here either, and consider (***i***) to be proven.

It is easy to prove that h satisfies property (***ii***):

$$
\begin{array}{l}
\quad h.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INL}.n)).[]) \\
= (\text{definition } h \text{ and } k) \\
\quad f_n.(\mathsf{OUTL}.(\mathsf{OUTR}.(\mathsf{INR}.(\mathsf{INL}.n)))) \\
= (\mathsf{OUTL}, \mathsf{OUTR}, \mathsf{INR}, \mathsf{INL}) \\
\quad f_n.n
\end{array}
$$

In order to show that h satisfies property (***iii***), we first need to prove:

$$
\forall s :: \mathsf{map}.(h \circ \mathsf{pick} \circ \mathsf{REP_Val}).(\mathsf{s2l}.s) = \mathsf{map}.g.(\mathsf{s2l}.s) \tag{4.2}
$$

proof of (4.2)
Let us consider an arbitrary set s of `Val`-typed values. Since $h \circ \mathsf{pick} \circ \mathsf{REP_Val}$ is mapped only on elements in $(\mathsf{s2l}\ s)$, it will be sufficient to prove:

$\forall t : t \in (\mathsf{s2l}\ s) : g = (h \circ \mathsf{pick} \circ \mathsf{REP_Val})$

From the definition of Q (3.3) and theorem 4.12, we know that:

$\forall x : x \in (\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP_Val}).(\mathsf{s2l}.s)) : (Q.x)$ holds,

and thus (Theorem B.10)

$\forall t : t \in (\mathsf{s2l}.s) : (Q.((\mathsf{pick} \circ \mathsf{REP_Val}).t))$

From (***i***) and (4.1) we can now deduce that

$\forall t : t \in (\mathsf{s2l}.s) : (g \circ \mathsf{ABS_Val} \circ \mathsf{equiv} \circ \mathsf{pick} \circ \mathsf{REP_Val}) = (h \circ \mathsf{pick} \circ \mathsf{REP_Val})$

which with theorem 4.2 and `Val`_ISO_DEF, rewrites to:

$\forall t : t \in (\mathsf{s2l}.s) : g = (h \circ \mathsf{pick} \circ \mathsf{REP_Val})$

end proof of (4.2)

Now we can proceed with the proof of property (***iii***) as follows:

$h.(\mathsf{Node}.(\mathsf{INL}.one).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP_Val}).(\mathsf{s2l}.s)))$
$= $ (definition h)
$k.(\mathsf{map}.h.(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP_Val}).(\mathsf{s2l}.s))).(\mathsf{INL}.one).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP_Val}).(\mathsf{s2l}.s))$
$= $ (map composition (Theorem B.8) and 4.2)
$k.(\mathsf{map}.g.(\mathsf{s2l}.s)).(\mathsf{INL}.one).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP_Val}).(\mathsf{s2l}.s))$
$= $ (definition k)
$f_s.(\mathsf{l2s}.(\mathsf{zip}.(\mathsf{map}.g.(\mathsf{s2l}.s), \mathsf{map}.(\mathsf{ABS_Val} \circ \mathsf{equiv}).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP_Val}).(\mathsf{s2l}.s)))))$
$= $ (map composition (Theorem B.8), theorem 4.2 and `Val`_ISO_DEF)
$f_s.(\mathsf{l2s}.(\mathsf{zip}.(\mathsf{map}.g.(\mathsf{s2l}.s), (\mathsf{s2l}.s))))$
$= $ (zip and split (Theorem B.9))
$f_s.(\mathsf{l2s}.(\mathsf{map}.(\mathsf{split}.g).(\mathsf{s2l}.s)))$
$= $ (l2s, map and image (Theorem C.9))
$f_s.(\mathsf{image}.(\mathsf{split}.g).(\mathsf{l2s}.(\mathsf{s2l}.s)))$
$= $ (s is a finite set (Theorem D.6))
$f_s.(\mathsf{image}.(\mathsf{split}.g).s)$

The proofs of properties (***iv***) and (***v***) are similar to the proof of (***iii***) and will not be given. We hereby finish the proof of the existence part of theorem 4.8, and continue with the proof that the existing paramorphism is unique. That is we shall prove that: for all function x and y of type `Val` $\rightarrow \gamma$:

$$\frac{\begin{array}{l}\forall n :: x.(\mathsf{NUM}.n) = f_n.n \\ \forall s :: \mathsf{finite}.s \Rightarrow (x.(\mathsf{SET}.s) = f_s.(\mathsf{image}.(\mathsf{split}.x).s)) \\ \forall l :: x.(\mathsf{LIST}.l) = f_l.(\mathsf{map}.(\mathsf{split}.x).l) \\ \forall t :: x.(\mathsf{TREE}.t) = f_t.(\mathsf{map_tree}.(\mathsf{split}.x).t) \\ \forall n :: y.(\mathsf{NUM}.n) = f_n.n \\ \forall s :: \mathsf{finite}.s \Rightarrow (y.(\mathsf{SET}.s) = f_s.(\mathsf{image}.(\mathsf{split}.y).s)) \\ \forall l :: y.(\mathsf{LIST}.l) = f_l.(\mathsf{map}.(\mathsf{split}.y).l) \\ \forall t :: y.(\mathsf{TREE}.t) = f_t.(\mathsf{map_tree}.(\mathsf{split}.y).t)\end{array}}{(x = y)} \tag{4.3}$$

In order to be able to prove this, we first need an induction theorem for type `Val`.

Theorem 4.16 INDUCTION ON `Val` *pvt_Induct*
For all properties H,

$$\frac{\begin{array}{l}\forall n :: H.(\mathsf{NUM}.n)\\ \forall s :: ((\mathsf{finite}.s) \wedge (\forall p :: (p \in s) \Rightarrow (H.p))) \Rightarrow (H.(\mathsf{SET}.s))\\ \forall l :: (\mathsf{every}.H.l) \Rightarrow (H.(\mathsf{LIST}.l))\\ \forall t :: (\mathsf{every_tree}.H.t) \Rightarrow (H.(\mathsf{TREE}.t))\end{array}}{\forall p :: H.p}$$

◀

The proof of this induction theorem is not too hard. Here we shall only give a sketchy proof to give the reader an idea. We start with the following lemma, that is easy to prove using `Val`_PROP.

Lemma 4.17 *induct_lemma4*
$(\forall p :: H.p) = (\forall t\, r :: ((r = \mathsf{equiv}.t) \wedge (Q.t)) \Rightarrow (H \circ \mathsf{ABS_Val} \circ \mathsf{equiv}).t)$

◀

To prove theorem 4.16 we assume:

$\mathbf{A}_1$) $\forall n :: H.(\mathsf{NUM}.n)$
$\mathbf{A}_2$) $\forall s :: ((\mathsf{finite}.s) \wedge (\forall p :: (p \in s) \Rightarrow (H.p))) \Rightarrow (H.(\mathsf{SET}.s)))$
$\mathbf{A}_3$) $\forall l :: (\mathsf{every}.H.l) \Rightarrow (H.(\mathsf{LIST}.l))$
$\mathbf{A}_4$) $\forall t :: (\mathsf{every_tree}.H.t) \Rightarrow (H.(\mathsf{TREE}.t))$

we have to prove that:

$(\forall p :: H.p)$
(= lemma 4.17)
$(\forall t\, r :: ((r = \mathsf{equiv}.t) \wedge (Q.t)) \Rightarrow (H \circ \mathsf{ABS_Val} \circ \mathsf{equiv}).t)$
($\Leftarrow$ `ltree` induction (Theorem E.4) and definition of every (Definition B.6))
for arbitrary h and tl, we have to prove:

$$\frac{\begin{array}{c}((r = \mathsf{equiv}.(\mathsf{Node}.h.tl)) \wedge (Q.(\mathsf{Node}.h.tl)))\\ \forall t :: (t \in tl) \Rightarrow \forall r :: ((r = \mathsf{equiv}.t) \wedge (Q.t)) \Rightarrow ((H \circ \mathsf{ABS_Val} \circ \mathsf{equiv}).t)\end{array}}{((H \circ \mathsf{ABS_Val} \circ \mathsf{equiv}).(\mathsf{Node}.h.tl))}$$

Moving the antecedents of this proof obligation into the assumptions, we get for an arbitrary h and tl that:

$\mathbf{A}_5$) $\forall t :: (t \in tl) \Rightarrow (\forall r :: ((r = \mathsf{equiv}.t) \wedge (Q.t)) \Rightarrow ((H \circ \mathsf{ABS_Val} \circ \mathsf{equiv}).t))$
$\mathbf{A}_6$) $r = \mathsf{equiv}.(\mathsf{Node}.h.tl)$
$\mathbf{A}_7$) $Q.(\mathsf{Node}.h.tl)$

The proof that $(H \circ \mathsf{ABS_Val} \circ \mathsf{equiv}).(\mathsf{Node}.h.tl)$, now proceeds by case distinction on h.

We shall prove the SET case (i.e. ISL.h), the other cases are similar. So suppose:

$\mathbf{A}_8$) ISL.h

From the definition of equiv, and the properties of Q, pick, ABS_Val and REP_Val it follows that:

Lemma 4.18 *SET_L2S_EQ_ABS*

For all lists tl of ((one + num + one + tree))ltrees:

$$\frac{\forall t :: (t \in tl) \Rightarrow (Q.t)}{(\mathsf{SET}.(\mathsf{l2s}.(\mathsf{map}.(\mathsf{ABS_Val} \circ \mathsf{equiv}).tl))) = (\mathsf{ABS_Val}.(\mathsf{equiv}.(\mathsf{Node}.(\mathsf{INL}.one).tl)))}$$

◀

Continuing with the proof of 4.16:

$(H \circ \mathsf{ABS_Val} \circ \mathsf{equiv}).(\mathsf{Node}.h.tl)$
$= (\mathbf{A}_8$, the type of h, one, and $\circ)$
$H.(\mathsf{ABS_Val}.(\mathsf{equiv}.(\mathsf{Node}.(\mathsf{INL}.one).tl)))$
$=$ (rewriting assumption $\mathbf{A}_7$ with Q, and lemma 4.18)
$H.(\mathsf{SET}.(\mathsf{l2s}.(\mathsf{map}.(\mathsf{ABS_Val} \circ \mathsf{equiv}).tl)))$
$\Leftarrow$ (assumption $\mathbf{A}_2$)
$\mathsf{finite}.(\mathsf{l2s}.(\mathsf{map}.(\mathsf{ABS_Val} \circ \mathsf{equiv}).tl))$
$\wedge$
$\forall p :: (p \in (\mathsf{l2s}.(\mathsf{map}.(\mathsf{ABS_Val} \circ \mathsf{equiv}).tl))) \Rightarrow (H.p)$
$=$ (lists are finite (Theorem C.10))
$\forall p :: (p \in (\mathsf{l2s}.(\mathsf{map}.(\mathsf{ABS_Val} \circ \mathsf{equiv}).tl))) \Rightarrow (H.p)$
$=$ (element of l2s and map (Theorems C.9, C.5 and C.11))
$\forall p :: (\exists t :: (t \in tl) \wedge (((\mathsf{ABS_Val} \circ \mathsf{equiv}).t) = p)) \Rightarrow (H.p)$

Making the antecedents of this proof obligation into assumptions, gives us an t, such that for arbitrary p:

$\mathbf{A}_9$) $t \in tl$
$\mathbf{A}_{10}$) $p = ((\mathsf{ABS_Val} \circ \mathsf{equiv}).t)$

leaving us with proof obligation:

$H.p$
$=$ (assumption $\mathbf{A}_{10}$)
$H.((\mathsf{ABS_Val} \circ \mathsf{equiv}).t)$
$\Leftarrow$ (Modus ponens assumption $\mathbf{A}_9$ and the Induction Hypothesis ($\mathbf{A}_5$))
$\exists r :: (r = \mathsf{equiv}.t) \wedge (Q.t)$
$=$ (rewriting assumption $\mathbf{A}_7$ with Q, and assumption $\mathbf{A}_9$)
$\exists r :: (r = \mathsf{equiv}.t)$

Instantiating with $\mathsf{equiv}\ t$ proves this case. As already indicated the other cases (where ISR h) are similar, the NUM case is trivial, and for the LIST and TREE cases, theorems similar to 4.18 had to be proved.

Now that an induction theorem on `Val` is available, it is straightforward to prove the uniqueness. Assume the premises of proof obligation (4.3). We have to prove:

$$
\begin{array}{l}
\quad x \ = \ y \\
= \text{(function equality)} \\
\quad \forall p :: (x.p) = (y.p) \\
\Leftarrow (\texttt{Val}\ \text{Induction},\ H = (\lambda p.\ (x.p) = (y.p))) \\
\quad \forall n :: \ (x.(\mathsf{NUM}.n)) = (y.(\mathsf{NUM}.n)) \\
\quad \wedge \\
\quad \forall s :: \ ((\mathsf{finite}.s) \ \wedge \ (\forall p :: \ (p \ \in \ s) \Rightarrow ((x.p) = (y.p))) \Rightarrow ((x.(\mathsf{SET}.s)) = (y.(\mathsf{SET}.s)))) \\
\quad \wedge \\
\quad \forall l :: \ (\mathsf{every}.(\lambda p.\ (x.p) = (y.p)).l) \Rightarrow ((x.(\mathsf{LIST}.l)) = (y.(\mathsf{LIST}.l))) \\
\quad \wedge \\
\quad \forall t :: \ (\mathsf{every_tree}.(\lambda p.\ (x.p) = (y.p)).t) \Rightarrow ((x.(\mathsf{TREE}.t)) = (y.(\mathsf{TREE}.t)))
\end{array}
$$

The first conjunct immediately follows from the premises of (4.3). We shall continue to prove the SET case, the LIST and TREE cases are similar. Suppose, for an arbitrary set s with `Val` typed values:

$$
\begin{array}{l}
\quad \mathbf{A'}_1)\ \mathsf{finite}.s \\
\quad \mathbf{A'}_2)\ \forall p :: \ (p \ \in \ s) \Rightarrow ((x.p) = (y.p))
\end{array}
$$

we have to prove that: $(x.(\mathsf{SET}.s)) = (y.(\mathsf{SET}.s))$. From the premises of (4.3), we can deduce that:

$$
\begin{array}{l}
\quad \mathbf{A'}_3)\ (x.(\mathsf{SET}.s)) = f_s.(\mathsf{image}.(\mathsf{split}.x).s) \\
\quad \mathbf{A'}_4)\ (y.(\mathsf{SET}.s)) = f_s.(\mathsf{image}.(\mathsf{split}.y).s) \\
\\
\quad (x.(\mathsf{SET}.s)) = (y.(\mathsf{SET}.s)) \\
= \text{(assumptions } \mathbf{A'}_3 \text{ and } \mathbf{A'}_4) \\
\quad f_s.(\mathsf{image}.(\mathsf{split}.x).s) = f_s.(\mathsf{image}.(\mathsf{split}.y).s) \\
\Leftarrow \\
\quad (\mathsf{image}.(\mathsf{split}.x).s) = (\mathsf{image}.(\mathsf{split}.y).s) \\
\Leftarrow \text{(Theorem C.6)} \\
\quad \forall p :: \ (p \ \in \ s) \Rightarrow ((\mathsf{split}.x.p) = (\mathsf{split}.y.p)) \\
= \text{(Definition 4.9 of } \mathsf{split}) \\
\quad \forall p :: \ (p \ \in \ s) \Rightarrow (((x.p), p) = ((y.p), p)) \\
= \text{(pairs)} \\
\quad \forall p :: \ (p \ \in \ s) \Rightarrow (x.p) = (y.p)
\end{array}
$$

Assumption $\mathbf{A'}_2$ proves this SET case, and, as indicated, the LIST and TREE cases are similar. This completes the outline of the uniqueness part, and consequently the entire proof of the abstract characterisation theorem of `Val` (Theorem 4.8).

5 Defining functions and operations on Val

In [Mee90] it is proved that all functions with source type σ are expressible in the form of a paramorphism, i.e. are paramorphisms. Consequently, functions on Val are defined in HOL as paramorphisms on Val. For example, a destructor function evaln, that unpacks the value (NUM n), is added to HOL as follows. First, using Hilbert's choice operator, evaln is defined as a function that has the desired behaviour:

Definition 5.1 evaln *evaln_DEF*

$\mathsf{evaln} = \varepsilon g.(\forall n.\ (g.(\mathsf{NUM}.n)\ =\ n))$

◀

Then we prove that the function evaln is a paramorphism:

Theorem 5.2 evaln IS A PARAMORPHISM ON Val *evaln*

$\forall n.\ \mathsf{evaln}.(\mathsf{NUM}.n)\ =\ n$

◀

proof of 5.2
For arbitrary n we have to prove that:
$\quad \mathsf{evaln}.(\mathsf{NUM}.n)\ =\ n$
$=$ (Definition 5.1)
$\quad (\varepsilon g.(\forall n.\ (g.(\mathsf{NUM}.n)\ =\ n))).(\mathsf{NUM}.n)\ =\ n$
$\Leftarrow$(Hilbert's ε (Theorem A.1))
$\quad \exists g.\ g.(\mathsf{NUM}.n)\ =\ n$
Specialising the conclusion of Theorem 4.8, by substituting $(\lambda n.\, n)$ for f_n, we can conclude the (unique) existence of a paramorphism para for which it holds that $\mathsf{para}.(\mathsf{NUM}.n)\ =\ n$. Now the existentially quantified proof obligation from above can be proved by reducing it with the witness para.
end proof of 5.2

Theorem 5.2 above is exactly desired the definition of the destructor function evaln. Note that is has been defined as a partial function by leaving the results of values not in the correct domain unspecified. If one wants to define evaln such that "undefinedness" is explicitly dealt with, first some constant of type Val has to be defined about which nothing can be proved, e.g.

```
new_constant {Name = "udef", Ty = ==`:Val`==} ;
```

Then evaln can be proved to be the paramorphism:

$\forall n.\ \mathsf{evaln}.(\mathsf{NUM}.n) = n$
$\forall set.\ (\mathsf{FINITE}.set) \Rightarrow \mathsf{evaln}.(\mathsf{SET}.set) = \mathtt{udef}$
$\forall l.\ \mathsf{evaln}.(\mathsf{LIST}.l) = \mathtt{udef}$
$\forall t.\ \mathsf{evaln}.(\mathsf{TREE}.t) = \mathtt{udef}$

by extending the **proof of** 5.2 with specialising the universally quantified functions f_{set}, f_l and f_t in Theorem 4.8 with functions that given an argument of the correct type return the value `udef`.

6 Conclusions

The contents of this report serves as a justification for adding the abstract characterisation theorem of the following data type as an axiom to the HOL theorem proving environment.

$$
\begin{array}{lcl}
\mathsf{VAL} & = & \mathsf{NUM}\ \texttt{num} \\
 & | & \mathsf{BOOL}\ \texttt{bool} \\
 & | & \mathsf{REAL}\ \texttt{real} \\
 & | & \mathsf{STR}\ \texttt{string} \\
 & | & \mathsf{SET}\ (\mathsf{VAL})\texttt{set} \\
 & | & \mathsf{LIST}\ (\mathsf{VAL})\texttt{list} \\
 & | & \mathsf{TREE}\ (\mathsf{VAL})\texttt{ltree}
\end{array}
$$

Although the data type `Val` is somewhat simpler than VAL, one can see that it contains the same problematic aspects as VAL (i.e. the constructors SET, LIST and TREE). From the verification activities described in this report, it becomes clear that manually adding the recursive data type VAL to HOL can be done analoguos to the way `Val` is added. The representation, abstract characterisation and proof obligations for the additional constructors BOOL, REAL and STRING, will be analogous to those of NUM. However, as the number of constructors increases the proofs become long and tedious. Since all formal proofs necessary to prove the abstract characterisation theorem of the subtype `Val` have been verified in HOL, we are convinced that the abstract characterisation theorem of VAL can also be proved. Therefore, we have added it to HOL as an axiom, saving time that was spent on proving theorems of which we were not yet convinced that they held.

7 Acknowledgements

We would like to thank Tom Melham, Graham Collins, and Lambert Meertens.

References

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL.* Cambridge University Press, 1993.

[Gor85] M.J.C. Gordon. HOL: A Machine Oriented Formulation of Higher Order Logic. Technical Report 68, University of Cambridge, Computer Laboratory, 1985.

[Har93] J. Harrison. Constructing the real numbers in HOL. In L.J.M. Claesen and M.J.C. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications (A-20)*, pages 145–164. Elsevier Science Publications BV North Holland, IFIP, 1993.

[Mee90] L. Meertens. Paramorphisms. Technical Report CS-R9005, CWI, Amsterdam, 1990.

[Mel89] T.F. Melham. Automating recursive type definitions in higher order logic. In P.A. Subrahmanyam and G. Birtwistle, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386. Springer-Verlag, 1989.

Appendices

A Hilbert's ε-operator

Hilbert's ε-operator is a primitive constant of higher order logic[Mel89, GM93]. Informally, its syntax and semantics are as follows. If $P[x]$ is a boolean term involving a variable x of type α, then $(\varepsilon x.\ P[x])$ denotes some value, say v, of type α such that $P[v]$ is true. If there is no such value (i.e. $\forall v \in \alpha : \neg P[v]$) then $(\varepsilon x.\ P[x])$ denotes some fixed but arbitrary chosen value of α[2]. For example:

- $\varepsilon n.\ 4 < n \wedge n < 6$ denotes the value 5
- $\varepsilon n.\ (\exists m :: n = 2 \times m)$ denotes an unspecified even natural number
- $\varepsilon n.\ n < n$ denotes an arbitrary natural number

The formalisation of the ε-operator in HOL is by the following theorem:

Theorem A.1 *SELECT_AX*

$$\forall P :: (\exists x :: P.x) \ \Rightarrow\ (P.(\varepsilon x.\ P.x))$$

Consequently, ε can be used to obtain a logical term which provably denotes a value with a given property P from a theorem merely stating the existence of such a value.

B Lists

In the built-in theory list, a type `('a)list` is defined to denote the set of all finite lists having elements of type `'a`. The constructor functions that are used to construct any list-structured value of type `('a)list` are:

$$[\,] \in \texttt{('a)list}$$
$$\mathsf{CONS} \in \texttt{'a} \rightarrow \texttt{('a)list} \rightarrow \texttt{('a)list}$$

Below definitions and theorems are stated. In these definitions, 0 and SUC refer to the constructor functions that are used to construct any natural number in `num`.

Theorem B.1 LIST ELEMENT *IS_EL*

$$(\forall x :: \neg\mathsf{is_el}.x.[]) \wedge (\forall x\ y\ l :: \mathsf{is_el}.y.(\mathsf{CONS}.x.l) = (y = x) \vee \mathsf{is_el}.y.l)$$

Throughout this technical report, when it is clear from the context that lists are used, is_el will be denoted by $\in$.

Definition B.2 MAP *MAP*

$$(\forall f :: \mathsf{map}.f.[] = []) \wedge (\forall f\ x\ l :: \mathsf{map}.f.(\mathsf{CONS}.x.l) = \mathsf{CONS}.(f\ x).(\mathsf{map}.f.l))$$

[2] A consequence, in HOL, types must be non-empty.

Definition B.3 ZIP *ZIP*

$(\mathsf{zip}.([],[]) \;=\; []) \wedge$
$(\forall x_1\, l_1\, x_2\, l_2 :: \mathsf{zip}.(\mathsf{CONS}.x_1.l_1, \mathsf{CONS}.x_2.l_2) \;=\; \mathsf{CONS}.(x_1, x_2).(\mathsf{zip}.(l_1, l_2)))$

Definition B.4 FOLDR *FOLDR*

$(\forall f\, e :: \mathsf{foldr}.f.e.[] \;=\; e) \;\wedge\; (\forall f\, e\, x\, l :: \mathsf{foldr}.f.e.(\mathsf{CONS}.x.l) \;=\; f.x.(\mathsf{foldr}.f.e.l))$

Definition B.5 LENGTH *LENGTH*

$(\mathsf{length}.[] \;=\; 0) \;\wedge\; (\forall x\, l :: \mathsf{length}.(\mathsf{CONS}.x.l) = (\mathsf{length}.l)) + 1$

Definition B.6 EVERY *EVERY*

$(\forall P :: \mathsf{every}.P.[] \;=\; \mathsf{true}) \;\wedge\; (\forall P\, h\, t :: \mathsf{every}.P.(\mathsf{CONS}.h.t) \;=\; P.h \;\wedge\; \mathsf{every}.P.t)$

Definition B.7 SUM *SUM*

$(\mathsf{sum}.[] \;=\; 0) \;\wedge\; (\forall x\, l :: \mathsf{sum}.(\mathsf{CONS}.x.l) \;=\; x + (\mathsf{sum}.l))$

Theorem B.8 MAP COMPOSTION *MAP_o*

$\forall f\, g\, l :: \mathsf{map}.f.(\mathsf{map}.g.l) \;=\; \mathsf{map}.(f \circ g).l$

Theorem B.9 *ZIP_MAP_EQ_MAP_split*

$\forall f\, l :: \mathsf{zip}.((\mathsf{map}.f.l), l) \;=\; \mathsf{map}.(\mathsf{split}.f).l$

Theorem B.10 *IS_EL_MAP*

$\forall Q\, f\, l :: (\forall x : \mathsf{is_el}.x.(\mathsf{map}.f.l) : Q.x) \;=\; (\forall x : \mathsf{is_el}.x.l : Q.(f.x))$

Definition B.11 *interval*

$\mathsf{interval}.0 = [] \;\wedge\; \mathsf{interval}.(\mathsf{SUC}.n) = \mathsf{CONS}.n.(\mathsf{interval}.n)$

Theorem B.12 APPENDING LISTS *APPEND*

$\forall l :: ([] +\!\!+ l) = l \;\wedge\; \forall l_1\, l_2\, x :: ((\mathsf{CONS}.x.l_1) +\!\!+ l_2) = (\mathsf{CONS}.x.(l_1 +\!\!+ l_2))$

Definition B.13 DELETING AN ELEMENT *DEL*

$(\forall y :: \mathsf{del}.y.[] = []) \wedge$
$(\forall y\, l\, x :: \mathsf{del}.y.(\mathsf{CONS}.x.l) = ((x = y) \rightarrow l \mid \mathsf{CONS}.x.(\mathsf{del}.y.l)))$

Definition B.14 FIRST ELEMENT OF A LIST *HD*

$\forall x\, l :: \mathsf{hd}.(\mathsf{CONS}.x.l) = x$

Definition B.15 TAIL OF A LIST *TL*

$\forall x\, l :: \mathsf{tl}.(\mathsf{CONS}.x.l) = l$

Definition B.16 INDEXED ELEMENTS *EL*

$\forall l :: \mathsf{el}.0.l = \mathsf{hd}.l \;\wedge\; \forall n\, l :: \mathsf{el}.(\mathsf{SUC}.n).l = \mathsf{el}.n.(\mathsf{tl}.l)$

Definition B.17 LIST CONTAINS NO DUPLICATE ELEMENTS *NO_DUPLICATES*

$\forall l :: \mathsf{no_duplicates}.l = \forall n\, k : n < \mathsf{length}.l \;\wedge\; k < \mathsf{length}.l \;\wedge\; n \neq k : \mathsf{el}.n.l \neq \mathsf{el}.k.l$

C Sets

Definition C.1 CHARACTERISTIC SET PREDICATE *IN_DEF*

$\forall s\, x ::\ \mathsf{CHF}.s.x\ =\ (x \in s)$

Definition C.2 IMAGE *IMAGE_DEF*

$\forall f\ s ::\ \mathsf{image}.f.s\ =\ \{f.x\ \mid x\ \in\ s\}$

Definition C.3 INSERT *INSERT_DEF*

$\forall x\ s ::\ x\ \mathsf{insert}\ s\ =\ \{y\ \mid\ (y = x)\ \vee\ y\ \in\ s\}$

Definition C.4 COMPLEMENT OF A SET

$\forall s :: s^{\mathsf{c}}\ =\ \{x \mid x \notin s\}$

Theorem C.5 *IN_IMAGE*

$\forall y\ s\ f ::\ y\ \in\ \mathsf{image}.f.s\ =\ (\exists x.\,(y\ =\ (f.x))\ \wedge\ x\ \in\ s)$

Theorem C.6 *IMAGE_EQ*

$$\forall f\ g\ s ::\ \frac{\forall x.\,(f.x) = (g.x)}{\mathsf{image}.f.s\ =\ \mathsf{image}.g.s}$$

The predicate FINITE is true of finite sets and false for infinite ones.

Definition C.7 CARDINALITY OF SETS *set*

$(\mathsf{card}.\{\} = 0)\ \wedge$
$\forall s ::\ \ \mathsf{FINITE}.s$
$\Rightarrow$
$(\forall x ::\ \mathsf{card}.(x\ \mathsf{insert}\ s) = ((x\ \in\ s) => \mathsf{card}.s\ \mid\ \mathsf{card}.s\ +\ 1))$

Definition C.8 CONVERTING LISTS TO SETS *L2S_DEF*

$(\mathsf{l2s}.[]\ =\ \{\})\ \wedge$
$(\forall x\ l ::\ \mathsf{l2s}.(\mathsf{CONS}.x.l)\ =\ x\ \mathsf{insert}\ (\mathsf{l2s}.l))$

Theorem C.9 *L2S_MAP_EQ_IMAGE*

$\forall f\ l ::\ \mathsf{l2s}.(\mathsf{map}.f.l)\ =\ \mathsf{image}.f.(\mathsf{l2s}.l)$

Theorem C.10 *L2S_FINITE*

$\forall l ::\ \mathsf{FINITE}.(\mathsf{l2s}.l)$

Theorem C.11 *IN_L2S_IS_EL*

$\forall l\ x ::\ (x\ \in\ (\mathsf{l2s}.l))\ =\ (\mathsf{is_el}.x.l)$

D Converting (finite) sets to lists

Because sets are unordered, a function that converts sets to lists *cannot* be defined by set-induction on finite sets, like e.g. the definition of function card:

$$
\begin{array}{ll}
\mathsf{s2l}.\{\} = [] \wedge & \\
\forall s :: & \mathsf{FINITE}.s \\
& \Rightarrow \\
& (\forall x :: \mathsf{s2l}.(x \ \mathsf{insert}\ s) = ((x \in s) => \mathsf{s2l}.s \mid \mathsf{CONS}.x.(\mathsf{s2l}.s)))
\end{array}
$$

Defining s2l this way, reults in that $\mathsf{s2l}.\{2,3,4\}$ is not equal to $\mathsf{s2l}.\{4,3,2\}$, and this is clearly not what we want.

The correct definition of the function s2l has been constructed as follows. First, we define when a function $f : \mathtt{num} \rightarrow \alpha$ is a bijection from a subset of num to a set containing elements of type α:

Definition D.1 Bijection from subset of num to a set *is_BIJ_NUM_to*

$$
\begin{array}{l}
f \ \mathsf{is_BIJ_NUM_to}\ s \\
\quad = \ \forall n\, m : n < \mathsf{card}.s \wedge m < \mathsf{card}.s : (f.n = f.m) \Rightarrow n = m \\
\qquad \wedge\ s = \{f.n \mid n < \mathsf{card}.s\}
\end{array}
$$

Then we define such a bijection for a set s using Hilbert's choice operator (Section A):

Definition D.2 A bijection for a set *set_BIJ*

$$\mathsf{set_BIJ}.s = \varepsilon f.\ f\ \mathsf{is_BIJ_NUM_to}\ s$$

and prove that it indeed is a bijection as defined in definition D.1:

Theorem D.3 *set_BIJ_DEF*

$$\forall s :: \mathsf{FINITE}.s \Rightarrow \mathsf{set_BIJ}.s\ \mathsf{is_BIJ_NUM_to}\ s$$

Now the idea is, to define the function $\mathsf{s2l}.s$ by mapping the set bijection $\mathsf{set_BIJ}.s$ as defined above on a list containing the elements 0 to $\mathsf{card}.s$.

Definition D.4 Converting finite sets to lists *S2L_L2S*

$$\mathsf{s2l}.s = \mathsf{map}.(\mathsf{set_BIJ}.s).(\mathsf{interval}.(\mathsf{card}.s))$$

The rest of this section lists some theorems.

Theorem D.5 *IN_IS_EL_S2L*

$$\forall s :: \mathsf{FINITE}.s \Rightarrow \forall x :: (x \in s) = (\mathsf{is_el}.x.(\mathsf{s2l}.s))$$

Theorem D.6 *L2S_S2L_id*

$$\forall s :: \mathsf{FINITE}.s \Rightarrow (s = \mathsf{l2s}.(\mathsf{s2l}.s))$$

Theorem D.7 *NO_DUPLICATES_S2L*

$$\forall s :: \mathsf{FINITE}.s \Rightarrow \mathsf{no_duplicates}.(\mathsf{s2l}.s)$$

Theorem D.8 *IS_EL_DEL_S2L*

$$\forall s\, x\, y \;::\; \frac{\mathsf{FINITE}.s \;\wedge\; \mathsf{is_el}.x.(\mathsf{s2l}.s) \;\wedge\; \mathsf{is_el}.y.(\mathsf{del}.x.(\mathsf{s2l}.s))}{x \neq y}$$

Theorem D.9 *S2L_split*

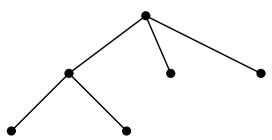
$$\forall s\, x \;::\; \frac{\mathsf{FINITE}.s \;\wedge\; \neg(x \in s)}{\exists l_1\, l_2 \;::\; (\mathsf{s2l}.(x\ \mathsf{insert}\ s) = (l_1 \mathbin{++} [x] \mathbin{++} l_2)) \;\wedge\; (s = \mathsf{l2s}.(l_1 \mathbin{++} l_2))}$$

E Labelled trees

In the built-in HOL theory tree, a type `tree` is defined to denote the set of all ordered trees of which the nodes can branch any (finite) number of times. A constructor function

$$\mathsf{node} \in (\mathtt{tree})\mathtt{list} \rightarrow \mathtt{tree}$$

can be used to construct any value of type tree. For example, the tree:



is denoted by the term:
$\mathsf{node}.[\mathsf{node}.[\mathsf{node}.[];\mathsf{node}.[]];\mathsf{node}.[];\mathsf{node}.[]]$

Figure 1: Some tree

The size of a tree is defined to be the number of nodes in that tree:

Definition E.1 SIZE OF TREES *size*

$\forall tl :: \mathsf{size}.(\mathsf{node}\ tl) = \mathsf{sum}.(\mathsf{map.size}.tl) + 1$

In the built-in HOL theory ltree a type of *labelled* trees (called `('a)ltree`) is defined by a type definition, following the way this is described in Section 2 and [Mel89]. Labelled trees have the same structure as values of the defined type `tree`. The only difference is that a labelled tree of type `('a)ltree` has a value associated with each of its nodes. The constructor:

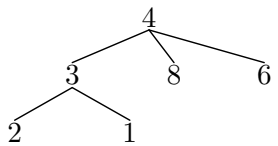
$$\mathsf{Node} \in \mathtt{'a} \rightarrow ((\mathtt{'a})\mathtt{ltree})\mathtt{list} \rightarrow (\mathtt{'a})\mathtt{ltree}$$

can be used to construct any value of type `('a)ltree`. For example,
There is a function available that given an labelled tree of type `('a)ltree` returns the shape of the tree:

$$\mathsf{shape}.t \in (\mathtt{'a})\mathtt{ltree} \rightarrow \mathtt{tree}$$

For example, applying shape to the labelled tree in Figure 2, returns the tree in Figure 1. The function that returns the list of values that are associated with the nodes of an labelled tree of type `('a)ltree` is:

$$\mathsf{values}.t \in (\mathtt{'a})\mathtt{ltree} \rightarrow (\mathtt{'a})\mathtt{list}$$



is denoted by the term:
$\mathsf{Node}.4.[\mathsf{Node}.3.[\mathsf{Node}.2.[];\mathsf{Node}.1.[]];\mathsf{Node}.6.[];\mathsf{Node}.8.[]]$

Figure 2: Some labelled tree

A pair $(t, l) \in$ (`tree` $\times$ (`'a`)`list`) for which it holds that the size of t equals the length of l can be used to create a labelled tree:

Definition E.2 IS_LTREE *Is_ltree*

$\forall t\ l ::\ \mathsf{Is_ltree}.(t, l) = (\mathsf{lenght}.l = \mathsf{size}.t)$

Theorem E.3 CAN CREATE LTREE FROM shape AND values *Is_ltree_REP_ltree_lemma*

$\forall t ::\ \mathsf{Is_ltree}.(\mathsf{shape}.t, \mathsf{values}.)$

◀

There is an induction principle on labelled trees:

Theorem E.4 LTREE INDUCTION *ltree_Induct*

$$\forall P ::\ \frac{(\forall t ::\ \mathsf{every}.P.t\ \Rightarrow\ (\forall h ::\ P.(\mathsf{Node}.h.t)))}{\forall l ::\ P.l}$$

◀

Finally, analoguous to the functions on lists, we have (defined in the theory more_ltrees):

Definition E.5 MAP ON TREES *MAP_TREE_DEF*

$\forall v\ t ::\ \mathsf{map_tree}.f.(\mathsf{Node}.v.t)\ =\ \mathsf{Node}.(f.v).(\mathsf{map}.(\mathsf{map_tree}.f).t)$

Definition E.6 ZIP ON TREES *ZIP_TREE_DEF*

For all v_1, v_2, t_1, t_2:

$$\frac{\mathsf{length}.t_1 = \mathsf{length}.t_2}{\mathsf{zip_tree}.(\mathsf{Node}.v_1.t_1, \mathsf{Node}.v_2.t_2)\ =\ \mathsf{Node}.(v_1, v_2).(\mathsf{map.zip_tree}.(\mathsf{zip}.(t_1, t_2)))}$$

Definition E.7 EVERY ON TREES *EVERY_TREE_DEF*

$(\forall P\ h\ t ::\ \mathsf{every_tree}.P.(\mathsf{Node}.h.t)\ =\ P.h\ \wedge\ \mathsf{every}.(\mathsf{every_tree}.P).t$

◀

Index

[ ] (constructor function for empty list), 19
++ (appending lists), 20
s^c (complement of set s), 21
ε (Hilbert's operator), 19

card, 21
CHF (characteristic set predicate), 21
concrete recursive types, 1
CONS (constructor function for lists), 19

define_new_type_bijections, 3
del, 20

el, 20
every, 20
every_tree, 24

FINITE, 21
foldr, 20
FST (HOL constant ('a#'b)→'a), 5

hd, 20
HOL
  type constants
    one, 4
  type operator
    list, 19
    ltree, 23
    tree, 23

image, 21
induction
  on labelled trees, 24
INL (HOL constant 'a→('a+'b)), 4
INR (HOL constant 'b→('a+'b)), 4
insert, 21
interval, 20
is_BIJ_NUM_to, 22
is_el, 19
Is_ltree, 24
ISL (HOL constant ('a+'b)→bool), 12
ISR (HOL constant ('a+'b)→bool), 12

l2s, 21
labelled trees induction principle, 24
length, 20
list (HOL type operator), 19
ltree (HOL type operator), 23
ltree (induction principle), 24

map, 19
map_tree, 24

new_type_definition, 2
no_duplicates, 20
Node (constructor function for ltree), 23
node (constructor function for tree), 23

one (HOL type constant), 4
OUTL (HOL constant ('a+'b)→'a), 12
OUTR (HOL constant ('a+'b)→'b), 12

paramorphism, 3

s2l, 22
shape (of ltrees), 23
size, 23
SND (HOL constant ('a#'b)→'b), 5
split, 10
SUC (successor function), 19
sum, 20

tl, 20
tree (HOL type operator), 23
type constant
  one, 4
type definition axiom, 2
type operator
  ltree, 23
  tree, 23

values (in ltree), 23

zip, 20
zip_tree, 24