Inductive data types with negative occurrences in HOL[*]

Tanja E.J. Vos
ITI, Universidad Politécnica de Valencia, tanja@iti.upv.es

S. Doaitse Swierstra
Informatica Instituut, Utrecht University, doaitse@cs.uu.nl

March 12, 2002

Abstract

We identify that a useful inductive data type *ty* with negative occurrences like $ty \rightarrow \mathsf{bool}$ in the arguments of its constructors can have a set-theoretic interpretation when the negative occurrence models only finite sets. Subsequently, we show how such data types can be manually added to higher order logic using equivalence sets.

1 Introduction

In theorem proving systems for higher order logics such as HOL [GM93], Isabelle [NPW02] and PVS [ORR$^+$96], tools are provided that automatically generate the theorems and definitions necessary to add inductive (or recursive) data types. Systems like PVS [OS93] use an axiomatic approach, i.e. the properties are generated syntactically only and introduced into the theory as axioms. In HOL [Mel89] and Isabelle [BW99] these tools use a definitional approach, meaning that the desired theorems are derived from the definitions within the system. However, not all inductive type descriptions have a solution in higher order logic and not all inductive type descriptions that have a solution can be automatically added to higher order logic [Gun93b]. In this paper we identify a useful inductive type of the latter kind and show how to manually add it to higher order logic.

Consider the following inductive type definition

$$ty = \mathsf{C}_1\ \tau_1^1\ \ldots\ \tau_1^{k_1}\ |\ \ldots\ |\ \mathsf{C}_m\ \tau_m^1\ \ldots\ \tau_m^{k_m} \tag{1.1}$$

Melham [Mel89] describes a set of tools in HOL that automatically carries out all the formal proofs necessary to add *concrete recursive types*, that is data types like (1.1) where each τ_i^j:

(1) is non-recursive, i.e. a type expression that does *not* include *ty* or

(2) is the name *ty* itself.

This is later on extended in [Mel91, Gun93a, Gun93b] to data types similar to the ones that can be automatically defined in Isabelle [BW99], i.e. data types like (1.1) where each τ_i^j should be an *admissible type definition*, meaning that each τ_i^j satisfies **(1)** or **(2)** or

(3) is of the form $(\tau'_1, \ldots, \tau'_n)t'$ where t' is an existing inductively defined data type[1] like (1.1) and $\tau'_1, \ldots, \tau'_n$ are admissible, or

---

[*] This work has been partially supported by CICYT grants TIC99-0280-C02-01 and TIC2000-1246.

[1] That means there exists an initiality theorem [Gun93b] (or paramorphism [Mee90]) for the (initial) data type

(4) is of the form $\sigma \to \tau'$, where τ' is an admissible type description and σ is non-recursive.

The last condition states that all occurrences of the newly defined type must be *strictly positive*. It is easy to see that violating this condition leads to inconsistencies [Gun93b]. For example, if one of the τ_i^j equals $ty \to \mathsf{bool}$ this would yield a contradiction since the cardinality of $ty \to \mathsf{bool}$ is that of the power-set of ty (i.e. $\mathcal{P}(ty)$) which by Cantor's theorem must be strictly greater than the cardinality of ty. However, this does not hold for $\mathcal{P}_{fin}(ty)$ – the subset of $\mathcal{P}(ty)$ that consists of only the finite sets – that, for infinite ty, has the same cardinality as ty. Consequently, we are able to assign a set-theoretic interpretation to a data type ty that has a τ_i^j equal to a "negative occurrence" like $ty \to \mathsf{bool}$, when this negative occurrence only represents the *finite* sets of ty. In this paper, we will show, using a definitional approach, how to manually add such a data type to HOL. More specific, we will describe how to define the following inductive data type Val:

$$\mathsf{Val} = \mathsf{SET}\ \mathsf{Val} \to \mathsf{bool} \mid \mathsf{NUM}\ \mathsf{num} \mid \mathsf{LIST}\ (\mathsf{Val})\mathsf{lists} \mid \mathsf{TREE}\ (\mathsf{Val})\mathsf{ltree} \qquad (1.2)$$

where arguments of the the constructor SET are restricted to be finite sets. This is somewhat smaller version of the data type used in [Vos00]:

$$\mathsf{Val} = \mathsf{SET}\ \mathsf{Val} \to \mathsf{bool} \mid \mathsf{NUM}\ \mathsf{num} \mid \mathsf{BOOL}\ \mathsf{bool} \mid \mathsf{REAL}\ \mathsf{real} \mid \mathsf{STR}\ \mathsf{string} \mid \mathsf{LIST}\ (\mathsf{Val})\mathsf{lists} \mid \mathsf{TREE}\ (\mathsf{Val})\mathsf{ltree}$$

to model the state-space of programs that have different variables taking different types (e.g. the Floyd-Warshall algorithm is an example of a program that needs variables of type set and variables of type number to solve the all-pairs shortest-path problem). In this paper we will not consider the booleans, reals and strings since these generate many proof obligations similar to those of the numerals.

2 Concepts and notation needed

This section gives a quick overview of the concepts we need in this paper. Function application is represented by a dot, function composition is defined as usual: $\forall f\ g :: f \circ g = (\lambda x.\ f.(g.x))$, and $\forall f\ x :: \mathsf{split}.f.x = (f.x, x)$. Hilbert's ε-operator in $(\epsilon x \cdot P.x)$, denotes some value, say v, such that $P.v$ holds. If there is no such value, a fixed but arbitrary value is returned. The type ('a)set is an abbreviation for the the type 'a→bool and elements of this type model finite sets if the predicate $\mathsf{finite}.s$ is true. For the type ('a)lists the empty list is denoted by [] and cons is the list constructor, and $\in$ is used for both set and list membership. We assume to have a function s2l that converts finite sets to lists, for its exact construction the reader is referred to [Vos00]. Moreover there is a function l2s that converts lists to sets. Theorems and definitions about sets and lists needed in this paper can be found in Appendix A. The type `tree` denotes ordered trees of which the nodes can branch any (finite) number of times. The size of a tree t (size.t) is defined to be the number of nodes in that tree. The function INL and INR are constructor functions for sum types, OUTR and OUTL project out of the right and left summand respectively, and ISL and ISR tests for membership of the left respectively right summand. Finally, fst and snd extract the first respectively second component of a pair (i.e. value of product type).

3 The general approach for defining a new type in HOL

Defining a new data type ty in HOL involves three steps [Gor85, Mel89]. First, find an appropriate *subset predicate* P of an existing type ety (the *representing type*) to represent the new type and show that P is not empty (i.e. $\exists x :: P.x$). Second, extend the syntax of logical types to include a new type symbol ty, and use the type definition axiom mechanism to add a definitional axiom to the logic asserting that the new type is isomorphic to the non-empty subset P of ety. The SML function `new_type_definition` {name = " *ty*", `pred` = P, `inhab_thm` = $\vdash \exists x :: P\ x$ } invoked in HOL, results in ty being a new type symbol characterised by the following definitional axiom:

$$\vdash \exists rep :: (\forall x\ y :: (rep.x = rep.y) \Rightarrow x = y) \wedge (\forall r :: \ P.r = (\exists x :: \ r = rep.x)) \qquad (ty\_\mathsf{TY\_DEF})$$

where *rep* can be thought of as a *representation function* that maps a value of the new type *ty* to the value of type *ety* that represents it. The type definition axiom (*ty*_TY_DEF) above, asserts only the *existence* of a bijection from *ty* to the corresponding subset of *ety*. To introduce constants that in fact denote this isomorphism and its inverse, we need to invoke:

```
define_new_type_bijections
        {ABS = "ABS_ty", REP = "REP_ty", name = "ty_ISO_DEF", tyax = ty_TY_DEF}
```

that defines REP_*ty*:*ty* → *ety* and ABS_*ty*:*ety* → *ty*, and creates the following theorem which is stored under the name *ty*_ISO_DEF:

$$\vdash (\forall a :: \ \mathsf{ABS}\_ty.(\mathsf{REP}\_ty.a) = a) \wedge (\forall r :: \ P.r = (\mathsf{REP}\_ty.(\mathsf{ABS}\_ty.r) = r)) \qquad ty\_\mathsf{ISO\_DEF}$$

It is straightforward to prove that these representation and abstraction functions are injective (one-to-one) and surjective (onto), using provided SML functions. Finally, stating that some property H is true for all elements of the new type *ty* is equal to stating that for all elements in P, H is true of their image under ABS_*ty*:

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

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 inductively defined data 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.

4 More concepts needed: labelled trees

Labelled trees, (`'a`)ltree, have the same structure as type `tree`, but also have a value associated with each of its nodes. They are is defined by a type definition using *existing type* (tree × (`'a`)lists) and *subset predicate* Is_ltree ⊆ (tree × (`'a`)lists) that is equal to the set of pairs $(t,l) \in$ (tree × (`'a`)lists) for which it holds that the size.t is equal to the length.l.

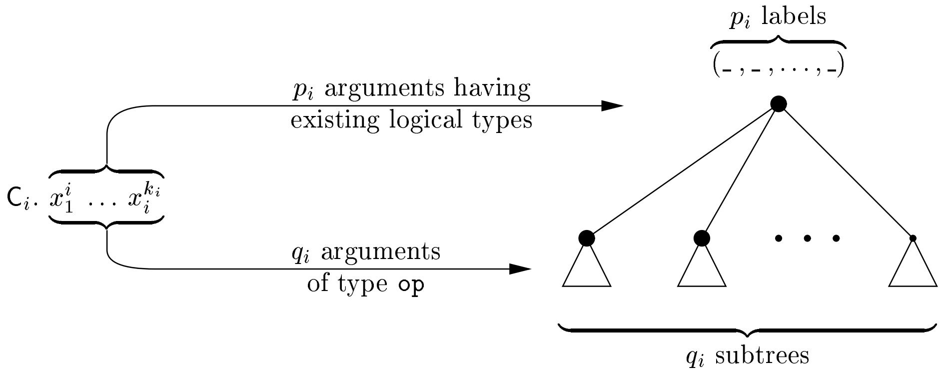
There is a function available that given an labelled tree of type (`'a`)ltree returns the shape of the tree: shape.t ∈ (`'a`)ltree → `tree`, defined by fst.(REP_ltree.t). The function that returns the list of values that are associated with the nodes: values.t ∈ (`'a`)ltree → (`'a`)lists is defined by snd.(REP_ltree.t).

The constructor: Node ∈ `'a` → ((`'a`)ltree)lists → (`'a`)ltree can be used to construct any value of type (`'a`)ltree. Some theorems we need in this paper can be found in Appendix A.

5 The representation and type definition

In [Mel89], each constructor $\mathsf{C}_i.x_i^1 \ \ldots \ x_i^{k_i}$ of a concrete recursive type like (1.1) is represented by a labelled tree. Suppose p_i is the number of arguments that have existing types and q_i is the number of arguments which have type op ($p_i + q_i = k_i$), then the abstract value of op denoted by

$\mathsf{C}_i.x_i^1 \ldots x_i^{k_i}$ is represented by a labelled tree that has p_i values associated with its root node, and q_i subtrees (for the recursive occurrences of op). In a diagram:

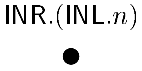


When $p_i = 0$, the representing tree is labelled with *one*, the one and only element of type one. When $q_i = 0$, the tree will have no subtrees. Each of the m constructors can be represented by a labelled tree in this way, and consequently the representing type for op will be:

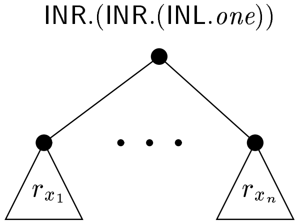
$$(\overbrace{\underbrace{(\_ \times \ldots \times \_)}_{\text{product of } p_1 \text{ types}} + \ldots + \underbrace{(\_ \times \ldots \times \_)}_{\text{product of } p_m \text{ types}}}^{\text{sum of } m \text{ products}})\mathsf{ltree}$$

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

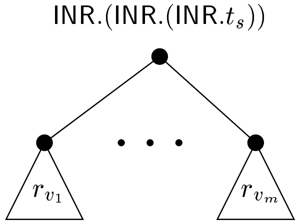
This method from [Mel89] only has to be adjusted a bit, in order to represent a subset of the new data type Val. Let us ignore the sets for a while, and start using the ideas outlined above. We use (one + num + one + tree)ltree as the representing type, and make representations for the constructors NUM,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



NUM.n



LIST.$[x_1, \ldots, x_n]$



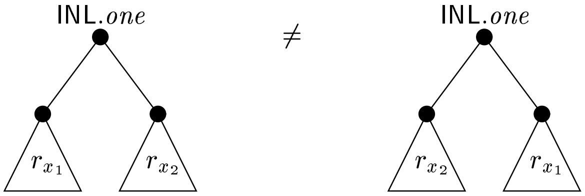
TREE.t
(where t = ABS_ltree$(t_s, [v_1, \ldots, v_n])$)

where, r_y denotes the representation as a (one + num + one + tree)ltree of value y of type Val, t_s is shape.t, and $[v_1, \ldots, v_m]$ is values.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.

Sets constitute 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 values of type Val by an equivalence class of ltrees in which ltrees like the two above are considered equivalent. Thus, the existing type used to represent our new type Val consists of equivalence classes of ltrees is: $(\mathsf{one} + \mathsf{num} + \mathsf{one} + \mathsf{tree})\mathsf{ltree} \to \mathsf{bool}$.

To define the subset predicate P, we need to formalise the equivalence relation equiv, that given an ltree of type $(\mathsf{one} + \mathsf{num} + \mathsf{one} + \mathsf{tree})\mathsf{ltree}$ returns the equivalence class of that ltree, i.e. $\mathsf{equiv} : (\mathsf{one} + \mathsf{num} + \mathsf{one} + \mathsf{tree})\mathsf{ltree} \to (\mathsf{one} + \mathsf{num} + \mathsf{one} + \mathsf{tree})\mathsf{ltree} \to \mathsf{bool}$. To formalise equiv we look at the equivalence classes of the possible values in Val:

$\mathsf{NUM}.n$ its equivalence class should contain only the ltree $(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INL}.n)).[])$. Consequently, $\mathsf{equiv}.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INL}.n)).[])$ must return a function that only delivers *true* for argument $(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INL}.n)).[])$.

$\mathsf{SET}.\{x_1, \ldots, x_n\}$ its equivalence class consist of ltrees equivalent to $\mathsf{Node}.(\mathsf{INL}.one).[r_{x_1}, \ldots, r_{x_n}]$, that is the class of ltrees that: (a) have $(\mathsf{INL}.one)$ at their root, and (b) 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 earlier are equivalent. Consequently, $\mathsf{equiv}.(\mathsf{Node}.(\mathsf{INL}.one).tl_1)$ must return a function that only delivers *true* when given an argument $(\mathsf{Node}.(\mathsf{INL}.one).tl_2)$ such that: $\mathsf{image}.\mathsf{equiv}.(\mathsf{l2s}.tl_1) = \mathsf{image}.\mathsf{equiv}.(\mathsf{l2s}.tl_2)$

$\mathsf{LIST}.[x_1, \ldots, x_n]$ its equivalence class consist of all ltrees that are present in the equivalence class of $\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INL}.one))).[r_{x_1}, \ldots, r_{x_n}]$, that is the class of ltrees that: (a) have the value $\mathsf{INR}.(\mathsf{INR}.(\mathsf{INL}.one))$ at their root, and (b) of which the **list** of images of their subtrees under equivalence are identical. Consequently, $\mathsf{equiv}.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INL}.one))).tl_1)$ must return a function that only delivers *true* for an argument $(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INL}.one))).tl_2)$ such that: $\mathsf{map}.\mathsf{equiv}.tl_1 = \mathsf{map}.\mathsf{equiv}.tl_2$

$\mathsf{TREE}.t$ – when t_s equals $\mathsf{shape}.t$ and $[v_1, \ldots, v_m]$ equals $\mathsf{values}.t$– its equivalence class consist of all ltrees that are present in the equivalence class of $\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.t_s))).[r_{v_1}, \ldots, r_{v_m}]$, that is the class of ltrees that: (a) have $\mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.t_s))$ at their root, and (b) of which the **list** of images of their subtrees under equivalence are identical. Consequently, invocation of $\mathsf{equiv}.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.t_s))).tl_1)$, 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: $\mathsf{map}.\mathsf{equiv}.tl_1 = \mathsf{map}.\mathsf{equiv}.tl_2$

Below the formal definition of equiv is given:

Def 5.1 EQUIVALENCE RELATION *equiv_DEF*

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

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:

Thm 5.2 $\mathsf{equiv}.t_1.t_2 \;=\; (\mathsf{equiv}.t_1 \;=\; \mathsf{equiv}.t_2)$ *equiv_EQUIV_REL*

The subset predicate P, specifying a non-empty subset of equivalence classes of ltrees, can now be defined as the quotient set of an appropriate subset Q of ltrees and the equivalence relation equiv. Looking at the representations of the different Val values, we can infer that this Q must satisfy:

Def 5.3 *Q_DEF*

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

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

Def 5.4 $P \;=\; Q/\mathsf{equiv}$ *Is_pvt_REP*

That means:

Thm 5.5 $P \;=\; (\lambda s.\, \exists t :: \; (s \;=\; \mathsf{equiv}.t) \wedge (Q.t))$ *Is_pvt_REP_THM*

It is easy to prove that P is not empty, and so we can use 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:

$\vdash (\forall a :: \; \mathsf{ABS\_Val}.(\mathsf{REP\_Val}.a) = a) \wedge (\forall r :: \; P.r = (\mathsf{REP\_Val}.(\mathsf{ABS\_Val}.r) = r))$	Val_ISO_DEF
$\vdash (\forall a\, a' :: \; (\mathsf{REP\_Val}.a = \mathsf{REP\_Val}.a') = (a = a'))$	Val_REP_ONE_ONE
$\vdash \forall r :: \; P.r = (\exists a :: \; r = \mathsf{REP\_Val}.a)$	Val_REP_ONTO
$\vdash \forall r\, r' :: \; P.r \Rightarrow P.r' \Rightarrow ((\mathsf{ABS\_Val}.r = \mathsf{ABS\_Val}.r') = (r \;=\; r'))$	Val_ABS_ONE_ONE
$\vdash \forall a :: \; \exists r :: \; (a \;=\; \mathsf{ABS\_Val}.r) \wedge P.r$	Val_ABS_ONTO
$\vdash (\forall x :: \; (H.x)) = (\forall r :: \; (P.r) \Rightarrow (H.(\mathsf{ABS\_Val}.r)))$	Val_PROP

6 The axiomatisation

The abstract axiomatisation of Val will be based upon the four constructors: NUM : num → Val, SET : (Val)set → Val, LIST : (Val)lists → Val, and TREE : (Val)ltree → Val. To define these 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:

Def 6.1 $\mathsf{pick}.c = \varepsilon t.\; c.t$ *pick*

It satisfies the following properties:

Thm 6.2 $\mathsf{equiv} \circ \mathsf{pick} \circ \mathsf{REP\_Val} \;=\; \mathsf{REP\_Val}$ *equiv_pick_REP_pvt*

Thm 6.3 $\forall x :: \; Q.((\mathsf{pick} \circ \mathsf{REP\_Val}).x)$ *Q_pick_REP_pvt*

Now the constructors can be defined as follows:

Def 6.4 *NUM_DEF, SET_DEF, LIST_DEF, TREE_DEF*

$$\begin{array}{l} \mathsf{NUM}.n = \mathsf{ABS\_Val}.(\mathsf{equiv}.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INL}.n)).[\,])) \\ \mathsf{SET}.s = \mathsf{ABS\_Val}.(\mathsf{equiv}.(\mathsf{Node}.(\mathsf{INL}.one).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP\_Val}).(\mathsf{s2l}.s)))) \\ \mathsf{LIST}.l = \mathsf{ABS\_Val}.(\mathsf{equiv}.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INL}.one))).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP\_Val}).l))) \\ \mathsf{TREE}.t = \mathsf{ABS\_Val}.(\mathsf{equiv}.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{shape}.t))))) \;.(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP\_Val}).(\mathsf{values}.t))) \end{array}$$

Having defined the constructors, the theorem which abstractly characterises the new type Val, by stating the unique existence of a paramorphism para has to be proved.

Thm 6.5 ABSTRACT CHARACTERISATION OF Val *pvt_Axiom*

$$\begin{array}{rll} \forall f_n\, f_s\, f_l\, f_t :: \exists!\mathsf{para} :: & & (\forall n :: \ \mathsf{para}.(\mathsf{NUM}.n) = f_n.n) \\ & \wedge & (\forall s :: \ (\mathsf{finite}.s) \Rightarrow (\mathsf{para}.(\mathsf{SET}.s) = f_s.(\mathsf{image}.(\mathsf{split}.\mathsf{para}).s)) \\ & \wedge & (\forall l :: \ \mathsf{para}.(\mathsf{LIST}.l) = f_l.(\mathsf{map}.(\mathsf{split}.\mathsf{para}).l)) \\ & \wedge & (\forall t :: \ \mathsf{para}.(\mathsf{TREE}.t) = f_t.(\mathsf{map\_tree}.(\mathsf{split}.\mathsf{para}).t)) \end{array}$$

The proof of Thm. 6.5 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 well-known theorem about quotient sets.

Thm 6.6 QUOTIENT SETS *QUOTIENT_THM*

For all equivalence relations E on α; Q that define a subset of α; $\mathsf{ABS} : (\alpha \rightarrow \mathsf{bool}) \rightarrow \beta$ and $\mathsf{REP} : \beta \rightarrow (\alpha \rightarrow \mathsf{bool})$, abstraction and representation functions respectively; and for all functions h of type $\alpha \rightarrow \gamma$:

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

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

Instantiating Thm. 6.6 with $(\mathsf{one} + \mathsf{num} + \mathsf{one} + \mathsf{tree})\mathsf{ltree}$ for α, equiv for E, $\mathsf{ABS\_Val}$ for ABS, $\mathsf{REP\_Val}$ for REP, and Q, obviously makes g a good candidate for para. Applying modus ponens to this instantiation and $\mathsf{Val\_ISO\_DEF}$ gives us a unique function g of type $\mathsf{Val} \rightarrow \gamma$ for which:

Lemma 6.7 $$\frac{\forall t_1\, t_2 :: \ (\mathsf{equiv}.t_1.t_2) \ \Rightarrow \ (h.t_1 = h.t_2)}{\forall t :: \ (Q.t) \ \Rightarrow \ (g.(\mathsf{ABS\_Val}.(\mathsf{equiv}.t)) = (h.t))}$$

Using Def. 5.3 of Q, Thm. 6.3 and A.17, it is easy to prove that:

Thm 6.8 *Q_NUM_REP, Q_SET_REP, Q_LIST_REP, Q_TREE_REP*

$$\begin{array}{l} \quad \forall n :: \ Q\ (\mathsf{Node}.(\mathsf{INR}.(\mathsf{INL}.n)).[]) \\ \wedge\ \forall s :: \ Q.(\mathsf{Node}.(\mathsf{INL}.one).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP\_Val}).(\mathsf{s2l}.s))) \\ \wedge\ \forall l :: Q.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INL}.one))).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP\_Val}).l)) \\ \wedge\ \forall t :: Q.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{shape}.t)))).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP\_Val}).(\mathsf{values}.t))) \end{array}$$

This together with Lemma 6.7 and Def. 6.4 of the constructors give us:

$$\frac{\forall t_1\, t_2 :: \ (\mathsf{equiv}.t_1.t_2) \ \Rightarrow \ (h.t_1 = h.t_2)}{\begin{array}{lcl} \forall n :: \ g.(\mathsf{NUM}.n) & = & h.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INL}.n)).[]) \\ \forall s :: \ g.(\mathsf{SET}.s) & = & h.(\mathsf{Node}.(\mathsf{INL}.one).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP\_Val}).(\mathsf{s2l}.s))) \\ \forall l :: \ g.(\mathsf{LIST}.l) & = & h.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INL}.one))).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP\_Val}).l)) \\ \forall t :: \ g.(\mathsf{TREE}.t) & = & h.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{shape.t})))).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP\_Val}).(\mathsf{values}.t))) \end{array}}$$

Consequently, to finish the existence part of the proof of Thm. 6.5 by reducing it with witness g, we have to find a function h, that satisfies the following properties for arbitrary f_n, f_s, f_l and f_t:

(***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***) $h.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INL}.one))).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP\_Val}).l)) = f_l.(\mathsf{map}.(\mathsf{split}.\mathsf{para}).l)$

(***v***) $h.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{INR}.(\mathsf{shape}.t)))).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP\_Val}).(\mathsf{values}.t))) = f_t.(\mathsf{map\_tree}.(\mathsf{split}.\mathsf{para}).t)$

The following function, defined by "primitive recursion" on ltrees, satisfies these conditions, and to finish the proof of the existence part of Thm. 6.5, it remains to validate this claim:

Def 6.9 *h_DEF*

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

To prove (***i***), we use the following theorem, the proof of which is straightforward but tedious since it involves lots of lemmas about zip.

Thm 6.10 *ltree_Axiom_PRESERVES_equiv*

For all functions h of type $(\mathsf{one} + \mathsf{num} + \mathsf{one} + \mathsf{tree})\mathsf{ltree} \to \gamma$ defined by “primitive recursion” on (one + num + one + tree)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 $(\gamma)\mathsf{lists} \to (\mathsf{one} + \mathsf{num} + \mathsf{one} + \mathsf{tree})\mathsf{lists} \to \gamma$):

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

To establish (***i***) we need to verify that our function k satisfies the premises of Thm. 6.10. Since this is again straightforward, we skip the proof and consider (***i***) as proved.

It is easy to prove that h satisfies property (***ii***) using Def. 6.9 and OUTL, OUTR, INR, INL, we get: $h.(\mathsf{Node}.(\mathsf{INR}.(\mathsf{INL}.n)).[]) = f_n.(\mathsf{OUTL}.(\mathsf{OUTR}.(\mathsf{INR}.(\mathsf{INL}.n)))) = f_n.n$.

To show that h satisfies property (***iii***), we first need to prove the following lemma:

Lemma 6.11 $\forall s :: \mathsf{map}.g.(\mathsf{s2l}.s)\ = \mathsf{map}.(h\ \circ\ \mathsf{pick} \circ\ \mathsf{REP\_Val}).(\mathsf{s2l}.s)$

Proof: For arbitrary s:
$(=$ A.7$)$ $\forall t : t \in (\mathsf{s2l}.s) : g.t = (h \circ \mathsf{pick} \circ\ \mathsf{REP\_Val}).t$
$(=$ Thm. 6.2 and Val_ISO_DEF$)$
$\forall t : t \in (\mathsf{s2l}.s) : (g \circ \mathsf{ABS\_Val} \circ \mathsf{equiv} \circ \mathsf{pick} \circ \mathsf{REP\_Val}).t = (h \circ \mathsf{pick} \circ \mathsf{REP\_Val}).t$
$\Leftarrow$ ((***i***) and Lemma 6.7) $\forall t : t \in (\mathsf{s2l}.s) : Q.((\mathsf{pick} \circ \mathsf{REP\_Val}).t)$
$(=$ A.5$)$ $\forall x : x \in (\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP\_Val}).(\mathsf{s2l}.s)) : (Q.x)$
this follows from Def. 5.3 and Thm. 6.8
end proof of 6.11

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

$$\begin{array}{l}
\ \ h.(\mathsf{Node}.(\mathsf{INL}.one).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP\_Val}).(\mathsf{s2l}.s))) \\
= (\text{Def. 6.9})\ k.(\mathsf{map}.h.(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP\_Val}).(\mathsf{s2l}.s))).(\mathsf{INL}.one).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP\_Val}).(\mathsf{s2l}.s)) \\
= (\text{A.6 and Lemma 6.11})\ k.(\mathsf{map}.g.(\mathsf{s2l}.s)).(\mathsf{INL}.one).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP\_Val}).(\mathsf{s2l}.s)) \\
= (\text{Def. 6.9})\ f_s.(\mathsf{l2s}.(\mathsf{zip}.(\mathsf{map}.g.(\mathsf{s2l}.s),\ \mathsf{map}.(\mathsf{ABS\_Val} \circ \mathsf{equiv}).(\mathsf{map}.(\mathsf{pick} \circ \mathsf{REP\_Val}).(\mathsf{s2l}.s))))) \\
= (\text{A.6, Thm. 6.2 and } \mathsf{Val\_ISO\_DEF})\ f_s.(\mathsf{l2s}.(\mathsf{zip}.(\mathsf{map}.g.(\mathsf{s2l}.s),\ (\mathsf{s2l}.s)))) \\
= (\mathsf{zip} \text{ and } \mathsf{split}\ (\text{A.8}))\ f_s.(\mathsf{l2s}.(\mathsf{map}.(\mathsf{split}.g).(\mathsf{s2l}.s))) \\
= (\mathsf{l2s}, \mathsf{map} \text{ and } \mathsf{image}\ (\text{A.14}))\ f_s.(\mathsf{image}.(\mathsf{split}.g).(\mathsf{l2s}.(\mathsf{s2l}.s))) \\
= (s \text{ is a finite set (A.15)}) \\
\qquad f_s.(\mathsf{image}.(\mathsf{split}.g).s)
\end{array}$$

The proofs of (***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 Thm. 6.5, and continue with the proof that the existing paramorphism is unique. That is we shall prove that:

Thm 6.12 For all functions x and y of type $\mathsf{Val} \to \gamma$:

$$\begin{array}{l}
\forall n ::\ (x.(\mathsf{NUM}.n) = f_n.n \wedge y.(\mathsf{NUM}.n) = f_n.n) \\
\forall s ::\ \mathsf{finite}.s \Rightarrow (x.(\mathsf{SET}.s) = f_s.(\mathsf{image}.(\mathsf{split}.x).s) \wedge y.(\mathsf{SET}.s) = f_s.(\mathsf{image}.(\mathsf{split}.y).s)) \\
\forall l ::\ (x.(\mathsf{LIST}.l) = f_l.(\mathsf{map}.(\mathsf{split}.x).l) \wedge y.(\mathsf{LIST}.l) = f_l.(\mathsf{map}.(\mathsf{split}.y).l)) \\
\forall t ::\ (x.(\mathsf{TREE}.t) = f_t.(\mathsf{map\_tree}.(\mathsf{split}.x).t) \wedge y.(\mathsf{TREE}.t) = f_t.(\mathsf{map\_tree}.(\mathsf{split}.y).t)) \\
\hline
\multicolumn{1}{c}{x = y}
\end{array}$$

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

Thm 6.13 Induction on Val *pvt_Induct*

$$\text{For all properties } H,\ \begin{array}{c}
(\forall n ::\ H.(\mathsf{NUM}.n)) \wedge (\forall s ::\ (\mathsf{finite}.s\ \wedge\ (\forall p ::\ p\ \in\ s \Rightarrow H.p)) \Rightarrow H.(\mathsf{SET}.s)) \\
(\forall l ::\ \mathsf{every}.H.l \Rightarrow H.(\mathsf{LIST}.l)) \wedge (\forall t ::\ \mathsf{every\_tree}.H.t \Rightarrow H.(\mathsf{TREE}.t)) \\
\hline
\forall p ::\ H.p
\end{array}$$

The proof of this induction theorem is not hard. Here we shall only give a sketchy proof to give the reader an idea (the HOL proof scripts are available upon request). We start with the following lemma, that is easy to prove using $\mathsf{Val\_PROP}$.

Lemma 6.14 $(\forall p ::\ H.p) = (\forall t\, r ::\ ((r = \mathsf{equiv}.t)\ \wedge\ (Q.t)) \Rightarrow (H \circ \mathsf{ABS\_Val} \circ \mathsf{equiv}).t)$

Continuing with the proof of Theorem 6.13 we assume:

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

so now we have to prove that:

$(\forall p ::\ H.p)$
$=$ (Lemma 6.14) $(\forall t\, r ::\ ((r = \mathsf{equiv}.t)\ \wedge\ (Q.t))\ \Rightarrow\ (H \circ \mathsf{ABS\_Val} \circ \mathsf{equiv}).t)$
$\Leftarrow$ (ltree induction (A.16) and definition of every (A.4))

For arbitrary h and tl, we have to prove:

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

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. $\mathsf{ISL}.h$), the other cases are similar. For the SET case we assume:

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

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

Lemma 6.15 *SET_L2S_EQ_ABS*

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

$$\begin{array}{c}
(\forall t ::\ t\ \in\ tl \Rightarrow Q.t) \\
\hline
(\mathsf{SET}.(\mathsf{l2s}.(\mathsf{map}.(\mathsf{ABS\_Val}\ \circ\ \mathsf{equiv}).tl))) = (\mathsf{ABS\_Val}.(\mathsf{equiv}.(\mathsf{Node}.(\mathsf{INL}.one).tl)))
\end{array}$$

Continuing with the proof of 6.13:

$(H\ \circ\ \mathsf{ABS\_Val}\ \circ\ \mathsf{equiv}).(\mathsf{Node}.h.tl)$
$=$ ($\mathbf{A}_8$, the type of h, *one*, and $\circ$) $H.(\mathsf{ABS\_Val}.(\mathsf{equiv}.(\mathsf{Node}.(\mathsf{INL}.one).tl)))$
$=$ (rewriting $\mathbf{A}_7$ with Q, and Lemma 6.15) $H.(\mathsf{SET}.(\mathsf{l2s}.(\mathsf{map}.(\mathsf{ABS\_Val}\ \circ\ \mathsf{equiv}).tl)))$
$\Leftarrow$ ($\mathbf{A}_2$ and lists are finite (A.13)) $\forall p ::\ (p\ \in\ (\mathsf{l2s}.(\mathsf{map}.(\mathsf{ABS\_Val}\ \circ\ \mathsf{equiv}).tl))) \Rightarrow (H.p)$

$=$ (element of $\mathsf{l2s}$ and map (A.14, A.10 and A.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 $\mathsf{ISR}\ h$) are similar, the NUM case is trivial, and for the LIST and TREE cases, theorems similar to 6.15 had to be proved.

Now that an induction theorem on Val is available (Thm. 6.13), it is straightforward to prove the uniqueness (i.e. Thm. 6.12). Assuming the premises of 6.12, we have to prove:
$x = y$
$=$ (function equality) $\forall p :: (x.p) = (y.p)$
$\Leftarrow$ (Val Induction, $H = (\lambda p.\ (x.p = y.p))$)
$\forall n :: (x.(\mathsf{NUM}.n) = y.(\mathsf{NUM}.n))$
$\wedge\ \forall s :: (\mathsf{finite}.s\ \wedge\ (\forall p :: p \in s \Rightarrow (x.p = y.p)) \Rightarrow (x.(\mathsf{SET}.s) = y.(\mathsf{SET}.s)))$
$\wedge\ \forall l :: (\mathsf{every}.(\lambda p.\ (x.p = y.p)).l) \Rightarrow (x.(\mathsf{LIST}.l) = y.(\mathsf{LIST}.l))$
$\wedge\ \forall t :: (\mathsf{every\_tree}.(\lambda p.\ (x.p = y.p)).t) \Rightarrow (x.(\mathsf{TREE}.t) = y.(\mathsf{TREE}.t))$
The first conjunct immediately follows from the premises of (6.12). We shall continue to prove the SET case, again the LIST and TREE cases are similar. Suppose, for an arbitrary set s with Val typed values:
$\mathbf{A'}_1$) $\mathsf{finite}.s \wedge \forall p :: p \in s \Rightarrow (x.p = y.p)$
From the premises of (6.12):
$\mathbf{A'}_2$) $(x.(\mathsf{SET}.s)) = f_s.(\mathsf{image}.(\mathsf{split}.x).s)$
$\mathbf{A'}_3$) $(y.(\mathsf{SET}.s)) = f_s.(\mathsf{image}.(\mathsf{split}.y).s)$
We have to prove that:
$x.(\mathsf{SET}.s) = y.(\mathsf{SET}.s)$
$=$ (assumptions $\mathbf{A'}_2$ and $\mathbf{A'}_3$) $f_s.(\mathsf{image}.(\mathsf{split}.x).s) = f_s.(\mathsf{image}.(\mathsf{split}.y).s)$
$\Leftarrow (\mathsf{image}.(\mathsf{split}.x).s) = (\mathsf{image}.(\mathsf{split}.y).s)$
$\Leftarrow$ (A.12) $\forall p :: p \in s \Rightarrow (\mathsf{split}.x.p) = (\mathsf{split}.y.p)$
$=$ (definition of split) $\forall p :: p \in s \Rightarrow ((x.p), p) = ((y.p), p)$
$=$ (pairs) $\forall p :: p \in s \Rightarrow x.p = y.p$
Assumption $\mathbf{A'}_1$ 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 (Thm. 6.5).

7 Concluding remarks and related work

We hope that this paper will help those that want to manually add inductive data types to HOL that do not fall exactly into the class of data types of the form (1.1) satisfying **(1)** till **(4)**, but that do have a sound set-theoretic interpretation.

Although in this paper we have concentrated mainly on the theorem prover HOL [GM93], our proofs are easily repeated within Isabelle [NPW02] since the latter contains the same type definition mechanism as HOL. Moreover, since we have verified the results in higher order logic using a definitional approach the results can be trusted, and hence can be added as axioms to a theorem prover like PVS that use axiomatic approaches.

All results in this paper have been verified with HOL (HOL90 version 7), the proof scripts are available from http://www.cs.uu.nl/~wishnu/research/hol_downloads/about.html or can be requested from the first author by sending an email.

Acknowledgments go to Tom Melham, Graham Collins, Lambert Meertens and Marieke Huisman.

References

[BW99] S. Berghofer and M. Wenzel. Inductive datatypes in HOL - lessons learned in formal-logic engineering. In *Theorem Proving in Higher Order Logics*, pages 19–36, 1999.

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

[Gor85] M.J.C. Gordon. HOL: A machine oriented formulation of higher order logic. Technical Report 68, University of Cambridge, Computer Laboratory, 1985.

[Gun93a] E.L. Gunter. A broader class of trees for recursive type definitions for HOL. In J.J Joyce and C.H Segers, editors, *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 780 of *LNCS*, pages 141–154. Springer-Verlag, Aug 1993.

[Gun93b] E.L. Gunter. Why we can't have `sml` style `datatype` declarations in `hol`. In L.J.M Claesen and M.J.C Gordon, editors, *Higher Order Logic Theorem Proving and its Applications*, pages 561–568. Elsevier Science Publications BV North Holland, 1993.

[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.

[Mel91] T.F. Melham. info-hol email 9 november, 1991.

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: The Tutorial*, volume 2283 of *LNCS*. Springer, 2002.

[ORR+96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: combining specifications, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *LNCS*, 1996.

[OS93] S. Owre and N. Shankar. Abstract datatypes in PVS. Technical Report CSL-93-9R, Computer Science Laboratory, SRI International, Menlo Park, CA, 1993.

[Vos00] T.E.J. Vos. *UNITY in Diversity, A stratified approach to the verification of distributed algorithms.* PhD thesis, Utrecht University (90-393-2316-X), 2000.

A Some definitions and theorems about lists, sets and trees

$$(\forall f :: \mathsf{map}.f.[] = []) \wedge (\forall f\,x\,l :: \mathsf{map}.f.(\mathsf{cons}.x.l) = \mathsf{cons}.(f\,x).(\mathsf{map}.f.l)) \tag{A.1}$$

$$(\mathsf{zip}.([],[]) = []) \wedge (\forall x_1\,l_1\,x_2\,l_2 :: \mathsf{zip}.(\mathsf{cons}.x_1.l_1, \mathsf{cons}.x_2.l_2) = \mathsf{cons}.(x_1,x_2).(\mathsf{zip}.(l_1,l_2))) \tag{A.2}$$

$$(\mathsf{length}.[] = 0) \wedge (\forall x\,l :: \mathsf{length}.(\mathsf{CONS}.x.l) = (\mathsf{length}.l)) + 1 \tag{A.3}$$

$$(\forall P :: \mathsf{every}.P.[] = \mathsf{true}) \wedge (\forall P\,h\,t :: \mathsf{every}.P.(\mathsf{cons}.h.t) = P.h \wedge \mathsf{every}.P.t) \tag{A.4}$$

$$\forall Q\ f\ l :: (\forall x : x \in (\mathsf{map}.f.l) : Q.x) = (\forall x : x \in l : Q.(f.x)) \tag{A.5}$$

$$\forall f\ g\ l :: \mathsf{map}.f.(\mathsf{map}.g.l) = \mathsf{map}.(f \circ g).l \tag{A.6}$$

$$\forall f\ g\ l :: (\forall x : x \in l : (f.x) = (g.x)) \Rightarrow (\mathsf{map}.f.l = \mathsf{map}.g.l) \tag{A.7}$$

$$\forall f\ l :: \mathsf{zip}.((\mathsf{map}.f.l), l) = \mathsf{map}.(\mathsf{split}.f).l \tag{A.8}$$

$$\forall f\ s :: \mathsf{image}.f.s = \{f.x \mid x \in s\} \tag{A.9}$$

$$\forall y\ s\ f :: y \in \mathsf{image}.f.s = (\exists x. (y = (f.x)) \wedge x \in s) \tag{A.10}$$

$$\forall l\ x :: (x \in (\mathsf{l2s}.l)) = (x \in l) \tag{A.11}$$

$$\forall f\ g\ s :: (\forall x. (f.x) = (g.x)) \Rightarrow (\mathsf{image}.f.s = \mathsf{image}.g.s) \tag{A.12}$$

$$\forall l :: \mathsf{finite}.(\mathsf{l2s}.l) \tag{A.13}$$

$$\forall f\ l :: \mathsf{l2s}.(\mathsf{map}.f.l) = \mathsf{image}.f.(\mathsf{l2s}.l) \tag{A.14}$$

$$\forall s :: \mathsf{finite}.s \Rightarrow (s = \mathsf{l2s}.(\mathsf{s2l}.s)) \tag{A.15}$$

$$\forall P :: (\forall t :: \mathsf{every}.P.t \Rightarrow (\forall h :: P.(\mathsf{Node}.h.t))) \Rightarrow (\forall l :: P.l) \tag{A.16}$$

$$\forall t :: \mathsf{Is\_ltree}.(\mathsf{shape}.t, \mathsf{values}.t) \tag{A.17}$$

$$\forall v\ t :: \mathsf{map\_tree}.f.(\mathsf{Node}.v.t) = \mathsf{Node}.(f.v).(\mathsf{map}.(\mathsf{map\_tree}.f).t) \tag{A.18}$$

$$\forall v_1, v_2, t_1, t_2 : \frac{\mathsf{length}.t_1 = \mathsf{length}.t_2}{\mathsf{zip\_tree}.(\mathsf{Node}.v_1.t_1, \mathsf{Node}.v_2.t_2) = \mathsf{Node}.(v_1, v_2).(\mathsf{map}.\mathsf{zip\_tree}.(\mathsf{zip}.(t_1, t_2)))} \tag{A.19}$$

$$\forall P\ h\ t :: \mathsf{every\_tree}.P.(\mathsf{Node}.h.t) = P.h \wedge \mathsf{every}.(\mathsf{every\_tree}.P).t \tag{A.20}$$