On the Representation of Datatypes in Isabelle/HOL

N. Völker

FernUniversität Hagen, Germany

Abstract

Representation of datatypes is a necessary prerequisite if one wants to prove rather than postulate the characteristic theorems of datatypes. This paper introduces two notions of representation functions for types and shows how representations of composed types can be calculated from representations of their constituents. Together with a representation of basic types due to Paulson [6], this provides a basis for the mechanization of datatypes in Isabelle/HOL.

0 Introduction

Datatypes are important ingredients of many theories modelling computations. We will be concerned here with datatypes which are generated from a number of elements and functions, the *constructors*. This means that every element of the type can be written as a constructor term, i.e. as an application of constructors to constructors. Furthermore, our datatypes will be freely generated by the constructors, i.e. the constructors are distinct and injective. This implies that every element of the datatype is denoted by a unique constructor term. As a consequence, such types enjoy a structural induction theorem and allow the definition of functions by primitive recursion.

In the categorical setting, these types arise as initial elements in certain categories of algebras ([2]). Therefore, one also speaks of initial algebras. Other names used are recursive or inductive types without laws. We will simply call them datatypes.

The dual kind of types are characterized by destructor functions whose domain is the carrier of the type. An example for such a "co-datatype" are infinite lists. Although parts of our discussion should carry over to co-datatypes, we will not consider them here.

Currently, the Isabelle/HOL system provides a datatype definition package which has a number of shortcomings:

1. The properties of the new types are postulated axiomatically.
2. The declaration of a type T may not contain applications of type operators to T. This forbids definitions like

$$('a, 'b)\ \texttt{Tree} \quad = \quad \texttt{LEAF}\ 'a \mid \texttt{NODE}\,('b, ('a, 'b)\ \texttt{Tree List}) \tag{1}$$

3. There is no support for mutually recursive datatypes.

We will outline in this paper an approach which aims to prove the characteristic theorems of the new datatype. This is achieved by representing types by sets of a certain type which was introduced by Paulson [6] for exactly this purpose. Our main contribution is the derivation of representations of type expressions from representations of

its constituents. This construction based on a generalization of the `map` function from lists to an arbitrary datatype.

The type $('a, 'b)\,\texttt{Tree}$ models trees with arbitrary finite branching and elements of $'a$ resp. $'b$ in the leafs resp. nodes. This example is inspired from a slightly simpler datatype in [1]. That article discusses some basic questions concerning automatic support for datatypes in another higher order proof assistant, namely the HOL system [5].

We are currently working on an implementation along the lines suggested in this paper. We stress that this paper reports on work in progress and does not claim to be the final word on representations of datatypes in higher order logic.

1 Map for arbitrary datatypes

At several places of our exposition, we will make use of the fact that for every datatype T one can define a function $T\_\texttt{map}$ which is the T analogue of the wellknown function `map` on lists. Since the definition of the general mapping function is technically somewhat involved, we will illustrate it first by a couple of examples. Recall that lists can be defined by the datatype declaration

$$'a\ \texttt{List} \;=\; [] \mid \texttt{Cons}\,('a,\ 'a\ \texttt{List}) \tag{2}$$

For the type $T = \texttt{List}$, the function $T\_\texttt{map}$ agrees with `map`, i.e. we have

$$\texttt{List\_map} \;:\; ('a \Rightarrow 'b) \Rightarrow ('a\ \texttt{List} \Rightarrow 'b\ \texttt{List})$$

and

$$\begin{array}{lcl} \texttt{List\_map}\ f\ [] & = & [] \\ \texttt{List\_map}\ f\ (\texttt{Cons}\,(x,\ xs)) & = & \texttt{Cons}\,(f\ x,\ \texttt{List\_map}\ f\ xs) \end{array}$$

Intuitively, `List_map` preserves the structure given by the list constructors $[]$ and `Cons`, but changes the values of those constructor arguments whose type in (2) is $'a$.

For the datatype `Tree` defined above, the type of the mapping function is:

$$\texttt{Tree\_map} \;:\; ['a_1 \Rightarrow 'b_1,\ 'a_2 \Rightarrow 'b_2] \Rightarrow (('a_1,\ 'a_2)\,\texttt{Tree} \Rightarrow ('b_1,\ 'b_2)\,\texttt{Tree})$$

Again, the function `Tree_map` is defined by primitive recursion. The recursive occurrence of `Tree` within `List` is reflected by an application of `List_map` to $\texttt{Tree\_map}\,(f_1, f_2)$:

$$\begin{array}{lcl} \texttt{Tree\_map}\,(f_1, f_2)\,(\texttt{LEAF}\ a_1) & = & \texttt{LEAF}\,(f_1\ a_1) \\ \texttt{Tree\_map}\,(f_1, f_2)\,(\texttt{NODE}\,(a_2, ts)) & = & \texttt{NODE}\,(f_2\ a_2, \texttt{List\_map}\,(\texttt{Tree\_map}\,(f_1, f_2))\ ts) \end{array}$$

For 0-ary datatypes, i.e. unparameterized types such as `nat`, the mapping function is the identity on that type. Next, we will show how to define the function $T\_\texttt{map}\ f$ for the case of an arbitrary one parameter datatype $'a\ T$ with m constructors. The declaration of such a type has the form

$$'a\ T \;=\; \texttt{C}_1(T_{1,1}, \ldots) \mid \ldots \mid \texttt{C}_m(T_{m,1}, \ldots) \tag{3}$$

where $T_{i,j}$ is the j'th argument type of constructor $\texttt{C}_i$. Of course, the number of type arguments can vary from constructor to constructor and can also be zero. Every $T_{i,j}$ is

a type expression build up from the type variable $'a$, recursive occurrences of $('a\ T)$ and previously defined datatypes. Note that T is only allowed to occur with the parameter $'a$, i.e. instantiations of T are not allowed.

Let f be some function from a type A to another type B. Then the mapping of f over T

$$T\_\mathtt{map}\ f \ : \ A\ T \Rightarrow B\ T$$

preserves constructors. For a constructor $\mathtt{C}_i$ with $k > 0$ arguments, we therefore have

$$T\_\mathtt{map}\ f\ (\mathtt{C}_i\,(x_{i,1}, \ldots,\ x_{i,k})) \ = \ \mathtt{C}_i\,(h\_T_{i,1}\ x_{i,1}, \ \ldots,\ h\_T_{i,k}\ x_{i,k})$$

for certain functions $h\_T_{i,j}$. On 0-ary constructors, the function $T\_\mathtt{map}\ f$ is the identity.

As indicated by the notation, the function $h\_T_{i,j}$ depends on the type $T_{i,j}$ in (3). For a type expression Ty, h_Ty is defined by induction on the structure of Ty as follows:

$$\begin{array}{lcl} h\_('a) & = & f \\ h\_('a\ T) & = & T\_\mathtt{map}\ f \\ h\_((T_1, \ldots, T_l)D) & = & D\_\mathtt{map}\,(h\_T_1, \ldots, h\_T_l) \end{array} \tag{4}$$

D stands here for an arbitrary previously datatype of some arity l.

For a general n-ary datatype T, the type of the mapping function is

$$T\_\mathtt{map} \ : \ ['a_1 \Rightarrow' b_1, \ldots, 'a_n \Rightarrow' b_n] \Rightarrow (('a_1, \ldots, 'a_n)\ T \ \Rightarrow \ ('b_1, \ldots, 'b_n)\ T)$$

The general definition of $T\_\mathtt{map}\,(f_1, \ldots, f_n)$ follows exactly the same scheme as for the case $n = 1$. The definition of the auxiliary functions $h\_T_{i,j}$ stays the same except that equation (4) is replaced by setting

$$h\_('a_i) \ = \ f_i,$$

for $i = 1, \ldots, n$.

Although it is not usual in higher order logic, the product and sum type themselves can be defined as datatypes:

$$\begin{array}{lcl} 'a\ +\ 'b & = & \mathtt{Inl}\,'a \ \mid \ \mathtt{Inr}\,'b \\ 'a\ *\ 'b & = & ('a, 'b) \end{array}$$

Their mapping operators are characterized by the following equations

$$\begin{array}{lcl} \mathtt{Sum\_map}\ (f, g)\ (\mathtt{Inl}\ a) & = & \mathtt{Inl}\ (f\ a) \\ \mathtt{Sum\_map}\ (f, g)\ (\mathtt{Inr}\ b) & = & \mathtt{Inr}\ (g\ b) \\ \mathtt{Prod\_map}\ (f, g)\ (a, b) & = & (f\ a,\ g\ b) \end{array} \tag{5}$$

A treatment of the generalized mapping function in the categorical framework can be found in [4].

2 Another generic function

For the definition of representing sets, we will be interested in computing the range of $T\_\mathtt{map}\,(f_1, \ldots, f_n)$. It turns out that this can be expressed in terms of another generic function

$$\mathtt{T\_set} \;:\; [{'a_1}\,\mathtt{set}, \ldots, {'a_n}\,\mathtt{set}] \;\Rightarrow\; ({'a_1}, \ldots, {'a_n})\; T\;\mathtt{set})$$

which can be defined for an arbitrary n-ary datatype T. Intuitively, $\mathtt{T\_set}\,(A_1, \ldots, A_n)$ will consist of those elements of T which can be generated from the sets $(A_1, \ldots, A_n)$. The meaning of 'generated' will be made precise using inductive definitions.

First, let us consider the example of the type `List`. The function

$$\mathtt{List\_set} \;:\; ({'a}\;\mathtt{set}) \;\Rightarrow\; (({'a}\;\mathtt{List})\;\mathtt{set})$$

should take a set A into the set of all lists with elements in A. This set is characterized by the following two introduction rules

$$\frac{}{[\,] \in \mathtt{List\_set}\;A} \qquad\qquad \frac{[a \in A;\;\; l \in \mathtt{List\_set}\;A]}{\mathtt{Cons}\,(a,\, l) \;:\; \mathtt{List\_set}\;A}$$

Note how each rule corresponds to the typing of one `List` constructor. In fact, we can derive every rule systematically from the typing rule of the corresponding constructor by simply replacing the name `List` by `List_set` and $'a$ by A.

For a general datatype $({'a_1}, \ldots, {'a_n})\; T$ with m constructors, we obtain analogously m introduction rules for $\mathtt{T\_set}\,(A_1, \ldots, A_n)$ from the typing rules of the constructors by replacing the name T by $T\_\mathtt{set}$ and $'a_i$ by A_i for $i \in [1, \ldots, n]$.

By the principle of inductive definition, a finite number of rules such as the ones above uniquely specifies a set, c.f. chapter 4 of [8] and the literature cited there. This set is the intersection of all the sets which comply to those rules.

Support for inductive definitions in higher order logic proof systems has been described in [3] and [6]. Both approaches implement inductively defined sets as least fixed points.

Our interest in $T\_\mathtt{set}$ stems from the equality:

$$\mathtt{range}\;(T\_\mathtt{map}\;(f_1, \ldots, f_n)) \;\;=\;\; T\_\mathtt{set}\;(\mathtt{range}\;f_1, \ldots, \mathtt{range}\;f_n) \qquad (6)$$

We note the fact that $T\_\mathtt{set}$ is obviously monotonic.

3 Representations of types

The principal means to add new types in Isabelle/HOL without risking inconsistencies is the `subtype` facility. This function is similar to the HOL systems `new_type_definition` and allows to define a new type T which is isomorphic to a nonempty subset S of an existing type RT. Isomorphic here means that there exists an injective function $T_{\mathtt{rep}}$ from the new type T onto its representing set S. In the `subtype` package, the new type is declared and constants are introduced for the representation function $T_{\mathtt{rep}}$ and an inverse abstraction function $T_{\mathtt{abs}}$. The inverse relationship between these two functions and the fact that the representing set S is the image of the representation function are postulated as axioms.

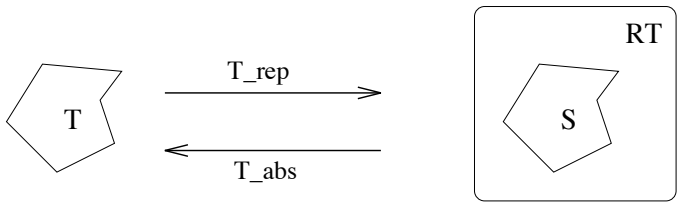

Figure 1: Definition of new types using `subtype`

The use of the `subtype` declaration is the only way to avoid the introduction of non trivial axioms in the definition of a datatype T in Isabelle/HOL. It implies the following subdivision of our task:

1. Construct a representing set S.
2. Define the new type T by a `subtype` declaration. This requires a proof that S is not empty.
3. Define the constructors by using the abstraction and representation functions.
4. Generate and prove the characteristic theorems.

In the following, we will be considering not just the representation of a single type but of a whole class $\mathcal{C}$ of types. Such a representation associates every type T in $\mathcal{C}$ with a representation function $T_{\texttt{rep}}$. Obviously, we require the representation functions to be injective.

Because we will need to combine representations, all the representation functions should ideally have the same target type `Rep`. However, there is a problem here. Consider the family $\mathcal{A}$ of types defined by

$$\mathcal{A}_n \quad = \quad ('a_1 \ * \ \ldots \ * \ 'a_n)$$

Clearly a type T_1 can only be embedded in another type T_2 if all type variables of T_1 also occur in T_2. This implies that it is not possible to embed all elements of $\mathcal{A}$ in one type. For a general class $\mathcal{C}$, the best we can ask for is that a type T parameterized by n different types variables $('a_1, \ldots, 'a_n)$ has a representation:

$$T_{\texttt{rep}} \ : \ ('a_1, \ldots, 'a_n)\ T \ \Rightarrow \ ('a_1 + \ldots +' a_n)\ \texttt{Rep} \tag{7}$$

Of course, repeated type parameters only have to occur once in the representation type. In particular, if all type parameters of a type are instantiated to one type parameter, we can expect a representation

$$T_{\texttt{rep}} \ : \ ('a, \ldots, 'a)\ T \Rightarrow \ 'a\ \mathit{Rep} \tag{8}$$

Lastly, unparameterized types T such as `nat` should be representable by functions

$$T_{\texttt{rep}} \ : \ T \ \Rightarrow \ 'a\ \texttt{Rep}$$

Our aim is now to find representations of type expressions built from types for which we already have a representation. As an example, consider the type

$$(('a,\ 'b)\ \texttt{Tree})\ \texttt{List}$$

which occurs in the definition of type `Tree`. Intuitively, a representation of a list *ts* of trees should be formed by first representing all elements of *ts* and then representing the resulting list of representations. This suggests a different notion of a representation of a parameterized type T, namely as a function which turns a T-structure of representations into a representation. Formally, this means that for a n-parameter datatype T we are looking for an injective function

$$T_{\texttt{REP}} \ :\ (\underbrace{'a\ \texttt{Rep}, \ldots, 'a\ \texttt{Rep}}_{n})\ T\ \Rightarrow\ 'a\ \texttt{Rep} \tag{9}$$

Given such a function, this would allow us to define

$$(('a,\ 'b)\ \texttt{Tree})\ \texttt{List}_{\texttt{rep}}\ =\ \texttt{List}_{\texttt{REP}} \circ \texttt{List\_map}\ \texttt{Tree}_{\texttt{rep}}$$

More generally, a representation of a type expression could be derived from representations of its constituents by setting

$$((T_1, \ldots, T_n)\ T)_{\texttt{rep}}\ =\ T_{\texttt{REP}} \circ T\texttt{\_map}\,(T_{1\,\texttt{rep}}, \ldots, T_{n\,\texttt{rep}}) \tag{10}$$

The injectivity of this function follows from the easily proven fact that the mapping Ty`_map` of injective functions over some type Ty is again injective.

Can we construct $T_{\texttt{REP}}$ from a given representation function $T_{\texttt{rep}}$? Instantiating (8) gives us

$$T_{\texttt{rep}}\ :\ ('a\ \texttt{Rep}, \ldots, 'a\ \texttt{Rep})\ \Rightarrow\ ('a\ \texttt{Rep})\ \texttt{Rep}$$

If we compare this with the type in (9), we note that the target type should be $'a$ `Rep`. This suggests that we require an injective function

$$\texttt{Rep\_flat}\ :\ ('a\ \texttt{Rep})\ \texttt{Rep}\ \Rightarrow\ 'a\ \texttt{Rep}$$

The name is borrowed from the function `flat` which flattens a list of lists by concatenation. Assuming the existence of such a function `Rep_flat`, we can simply define the desired function $T_{\texttt{REP}}$ by

$$T_{\texttt{REP}}\ =\ \texttt{Rep\_flat} \circ T_{\texttt{rep}}$$

The injectivity of $T_{\texttt{REP}}$ follows of course from the fact that the composition of injective functions is injective.

Unfortunately, there are types `Rep` which we might want to use as a representation type, but for which no such injection `Rep_flat` exists. A prominent example is the type of endomorphic functions

$$'a\ \texttt{Fun}_1\ =\ ('a\ \Rightarrow\ 'a)$$

Its 'flattening' function is of type

$$\texttt{Fun}_1\texttt{\_flat}\ :\ (('a\ \Rightarrow\ 'a)\ \Rightarrow\ ('a\ \Rightarrow\ 'a))\ \Rightarrow\ ('a\ \Rightarrow\ 'a)$$

Now, for arbitrary types T_1 and T_2, the HOL type $T_1 \Rightarrow T_2$ contains all set theoretic functions from T_1 to T_2. However, it is wellknown in set theory that only one-element sets S allow an injection of type $(S \Rightarrow S) \Rightarrow S$. This follows from the fact that for S with a cardinality strictly greater than one the cardinality of $S \Rightarrow S$ is always greater than the cardinality of S.

Hence there exists no injective function $\texttt{Fun}_1\texttt{\_flat}$ of the type required above. The same argument also shows immediately that the function space operator has no _REP representation. This is in contrast to the fact that it is easy to construct a representation

$$\texttt{Fun}_{\texttt{rep}} \;:\; ('a \Rightarrow 'b) \Rightarrow ('a + 'b)\ \texttt{Fun}_1$$

After constructing $T_{\texttt{REP}}$ from $T_{\texttt{rep}}$, assume now conversely that we are given a representation function $T_{\texttt{REP}}$ of type (9). Let

$$\texttt{In}_{n,i} \;:\; 'a_i \Rightarrow 'a_1 + \ldots + 'a_n$$

be the i'th injection into the n-fold sum. Further, assume the existence of an embedding

$$\tau \;:\; 'a \Rightarrow 'a\ \texttt{Rep}$$

Then we can define a representation function $T_{\texttt{rep}}$ of type (7) by setting

$$T_{\texttt{rep}} \;=\; T_{\texttt{REP}} \circ T\texttt{\_map}\ (\tau \circ \texttt{In}_{n,1}, \ldots, \tau \circ \texttt{In}_{n,n})$$

The two kinds of representation functions are therefore equivalent in the sense that we can calculate one from the other under suitable assumptions about the representation type Rep. Note that the two notions coincide for types without constants.

The above section describes only some basic requirements of type representations. We are currently investigating the usefulness of further properties like monad axioms for Rep, c.f. [7].

4 Representation of some basic types

For the derivation of representing sets for datatypes, we require at least representations of a number of basic types. These types are the polymorphic type $'a$, the unit type unit and the sum and product types. The necessity to represent these types stems from the fact that semantically the bars and commas in a datatype declaration correspond to the sum resp. product of types. The constructor names are simply the tags of the sum. An empty argument list corresponds to the unit type. The representation of $'a$ is necessary for the treatment of type variables.

A construction of representations for these four types has been given by L. Paulson in [6]. It was implemented by him in the theory Univ which is part of the standard Isabelle-94 HOL library. Rather than listing this theory here, we will just summarize some of its main features. Details can be found in the cited paper resp. in the theory files.

In Univ.thy, the type used for representation is called item. It corresponds to the type Rep of the previous section. We will not go into the definition of this type

itself. For our discussion here, the main point of the type `item` is the existence of three functions

$$\begin{array}{lcl}
\texttt{Leaf} & : & 'a \Rightarrow 'a\ \texttt{item} \\
\texttt{Numb} & : & \texttt{nat} \Rightarrow 'a\ \texttt{item} \\
\$ & : & ['a\ \texttt{item}, 'a\ \texttt{item}] \Rightarrow 'a\ \texttt{item} \qquad (\texttt{infix})
\end{array}$$

which are injective and have disjoint ranges. Hence these functions generate a subset of `'a item` which is isomorphic to a datatype with these three functions as constructors.

Function `Leaf` corresponds to the function τ from above. It provides a representation of $'a$. More generally, n different type variables $'a_1, \ldots, 'a_n$ are represented by setting

$$'a_{i\,\texttt{rep}} \;=\; \texttt{Leaf} \circ \texttt{In}_{n,i}$$

for $i \in [1, \ldots, n]$.

Function `Numb` provides a representation of natural numbers. In particular, this can be used to define a representation of the unit type by

$$\texttt{unit}_{\texttt{REP}} \;=\; \texttt{Numb}(0)$$

Using the bijection

$$\lambda\ f\ ab.f\ (\texttt{fst}\ ab)\ (\texttt{snd}\ ab)$$

between two-parameter functions and functions with pairs as parameters, we can transform operator \$ into a `REP`- representation of the product type:

$$\begin{array}{lcll}
\texttt{Prod}_{\texttt{REP}} & = & 'a\,\texttt{item} * 'a\,\texttt{item} \Rightarrow 'a\,\texttt{item} & \\
\texttt{Prod}_{\texttt{REP}} & = & \lambda\ ab.(\texttt{fst}\ ab)\ \$\ (\texttt{snd}\ ab) & (11)
\end{array}$$

For the definition of a representation of $+$, injections

$$\begin{array}{lcl}
\texttt{In0}(A) & \equiv & \texttt{Numb}(0)\ \$\ A \\
\texttt{In1}(A) & \equiv & \texttt{Numb}(1)\ \$\ A
\end{array}$$

are defined. Since these functions have disjoint images, one obtains the following `REP`-representation of the sum type

$$\begin{array}{lcl}
\texttt{Sum}_{\texttt{REP}} & = & 'a\,\texttt{item} + 'a\,\texttt{item} \Rightarrow 'a\,\texttt{item} \\
\texttt{Sum}_{\texttt{REP}} & = & \texttt{sum_case}\ \texttt{In0}\ \texttt{In1}
\end{array}$$

For the representation of type expressions, it is important that we can define an injection:

$$\texttt{item\_flat} \;:\; ('a\ \texttt{item})\ \texttt{item} \Rightarrow 'a\ \texttt{item}$$

Its effect on the subset of `'a item` spanned by the three functions `Leaf`, `Numb` and \$ is given by the primitive recursive equalities

$$\begin{array}{lcl}
\texttt{item_flat}\ (\texttt{Leaf}\ a) & = & a \\
\texttt{item_flat}\ (\texttt{Numb}\ n) & = & \texttt{Numb}\ n \\
\texttt{item_flat}\ (M\ \$\ N) & = & \texttt{item_flat}\,M\ \$\ \texttt{item_flat}N
\end{array}$$

For the derivation of `map` functions on datatypes, an `item_map` function is useful. Again, we do not give its definition but only note the following equalities:

$$\begin{array}{lcl} \texttt{item\_map}\ f\ (\texttt{Leaf}\ a) & = & \texttt{Leaf}\ (f\ a) \\ \texttt{item\_map}\ f\ (\texttt{Numb}\ n) & = & \texttt{Numb}\ n \\ \texttt{item\_map}\ f\ (M\ \$\ N) & = & \texttt{item\_map}\, f\, M\ \$\ \texttt{item\_map}\, f\, N \end{array}$$

The two functions `item_flat` and `item_map` are at the time of this writing not part of the standard Isabelle distribution.

Because of the injectivity of representation functions, the range of $T_{\texttt{rep}}$ can serve as a representing set for a type T. For the following range calculations, note the equality

$$\texttt{range}\ (f \circ g) \quad = \quad f\ \texttt{"}\ (\texttt{range g}) \tag{12}$$

It uses the image operator " defined by

$$f\ \texttt{"}\ A \quad = \quad \bigcup\ a \in A.\ \{f\ a\}$$

The range of a general type instantiation can be computed as follows:

$$\begin{array}{l} \texttt{range}\ ((T_1, \ldots, T_n)\ T)_{\texttt{rep}} \\ \qquad = \{\ \text{equation (10)}\ \} \\ T_{\texttt{REP}}\ \texttt{"}\ \texttt{range}\ (T_{\texttt{REP}} \circ T\texttt{\_map}\, (T_{1\,\texttt{rep}}, \ldots, T_{n\,\texttt{rep}})) \\ \qquad = \{\ \text{equations (12,6)}\ \} \\ T_{\texttt{REP}}\ \texttt{"}\ T\texttt{\_set}\ (\texttt{range}\ T_{1\,\texttt{rep}}, \ldots, \texttt{range}\ T_{n\,\texttt{rep}}) \end{array}$$

This expression can be evaluated further for concrete T. In the case of the sum type, we have

$$\texttt{Prod\_set}\ (A, B) \quad = \quad \bigcup a \in A.\ \bigcup b \in B\ .\ \{(a, b)\} \tag{13}$$

which implies

$$\begin{array}{l} \texttt{range}\ (A\ *\ B)_{\texttt{rep}} \\ \qquad = \{\ \text{derivation above}\ \} \\ \texttt{Prod}_{\texttt{REP}}\ \texttt{"}\ (\texttt{Prod\_set}\ (\texttt{range}\, A_{\texttt{rep}}, \texttt{range}\, B_{\texttt{rep}})) \\ \qquad = \{\ \text{equations (13,11), simplifications}\ \} \\ \bigcup\ a \in \texttt{range}\ A_{\texttt{rep}}\ .\ \bigcup\ b \in \texttt{range}\ B_{\texttt{rep}}\ .\{a\ \$\ b\} \\ \qquad = \{\ \text{definition of <*> below}\ \} \\ \texttt{range}\ A_{\texttt{rep}}\ \texttt{<*>}\ \texttt{range}\ B_{\texttt{rep}} \end{array}$$

where the binary operator <*> is defined by:

$$X\ \texttt{<*>}\ Y \quad = \quad \bigcup\ a \in A\ .\ \bigcup\ b \in B\ .\ \{a\ \$\ b\}$$

Analogously, we can derive:

$$\texttt{range}\ (A\ +\ B)_{\texttt{rep}} \quad = \quad \texttt{range}\ A_{\texttt{rep}}\ \texttt{<+>}\ \texttt{range}\ B_{\texttt{rep}}$$

where the binary operator <+> is given by:

$$X\ \texttt{<+>}\ Y \quad = \quad \texttt{In0}\ \texttt{"}\ X \cup \texttt{In1}\ \texttt{"}\ Y$$

Note that the two operators <*> and <+> are obviously monotonic.

5 The representing set of a datatype

We will now explain how the little representation theory of the preceding sections can be used to construct representing sets for datatypes. Instead of showing this abstractly, we will demonstrate it for the case of the type `Tree` and trust that the reader can infer the general procedure. Recall the definition of `Tree`:

$$('a, 'b)\ \texttt{Tree} \;=\; \texttt{LEAF}\ 'a \mid \texttt{NODE}\,('b, ('a, 'b)\ \texttt{Tree List})$$

As mentioned before, the bars and the commas in a datatype declaration correspond to sum and product of types. This means that we can interpret this type declaration as an isomorphism of types

$$('a, 'b)\ \texttt{Tree} \;\equiv\; 'a + 'b * ('a, 'b)\ \texttt{Tree List} \tag{14}$$

This isomorphism suggests that the range of the representation function of both sides should be the same, i.e.

$$\texttt{range}\ (('a, 'b)\ \texttt{Tree})_{\texttt{rep}} \;=\; \texttt{range}\ ('a + 'b * ('a, 'b)\ \texttt{Tree List})_{\texttt{rep}} \tag{15}$$

Let

$$R \;=\; \texttt{range}\,(('a, 'b)\ \texttt{Tree}_{\texttt{rep}})$$

Using the representation calculus developed in the previous sections, equation (15) can be reduced to:

$$\begin{array}{rcl} R & = & \texttt{range}\,(\texttt{Leaf} \circ \texttt{Inl}) \\ & & <+>\ (\texttt{range}\,(\texttt{Leaf} \circ \texttt{Inr})\ <*>\ \texttt{List}_{\texttt{REP}}\ ``\ \texttt{List\_set}\ R) \end{array}$$

Because of the monotonicity of the occurring operations on set, the abstraction of the right hand side over R is a monotonous function. Hence Tarski's fixed point theorem implies the existence and uniqueness of a smallest solution R to this equality, c.f. [6] or the theory `Lfp` in the standard Isabelle/HOL distribution. This set is used as the representing set of $('a, 'b)$ `Tree`.

In general, the construction of the representing set requires that we already have representation functions of all the types occurring in the definition of a datatype T except of course for T itself.

Instead of the definition of R as a least fixed point, an equivalent characterization using inductive sets is also possible. In our example, we get the following two rules

$$\frac{}{\texttt{In0}\,(\texttt{Leaf}\,(\texttt{Inl}\ a)) \in \texttt{R}} \qquad\qquad \frac{rs \in \texttt{List\_set R}}{\texttt{In1}\,(\texttt{Leaf}\,(\texttt{Inr}\ b)\ \$\ \texttt{List}_{\texttt{REP}}\ rs)}$$

In general, if there are more than two constructors, it is necessary to distinguish them by suitable combinations of `In0` and `In1`.

6 Definition of type and constructors

According to our work-plan from section 3, we can now declare the type `Tree` by a `subtype` definition using R as representing set. The non-emptiness of R follows easily from equality (**??**). For a general datatype T, it requires at least one constructor whose argument types do not contain recursive occurrences of T.

From the `subset` declaration of `Tree`, we obtain a representation function $\texttt{Tree}_{\texttt{rep}}$ and an inverse abstraction function $\texttt{Tree}_{\texttt{abs}}$. The typing of $\texttt{Tree}_{\texttt{rep}}$ agrees with the general typing (7). Per construction, the ranges of the representing functions of the two types in (14) are equal. Note that nothing is said about the equality of the representation functions themselves.

The constructors of a new type T are obtained by

1. representing the arguments,
2. combining the results using \$ and `In0`/`In1` and
3. abstracting into T.

In the case of type `Tree`, this results in

$$
\begin{array}{lcl}
\texttt{LEAF} & : & 'a \Rightarrow ('a, 'b)\ \texttt{Tree} \\
\texttt{NODE} & : & ['b, ('a, 'b)\ \texttt{Tree}] \Rightarrow ('a, 'b)\ \texttt{Tree} \\
\texttt{LEAF}\ a & = & \texttt{Tree}_{\texttt{abs}}\ (\texttt{In0}\ (\texttt{Leaf}\ (\texttt{Inl}\ a))) \\
\texttt{NODE}\ (b, ts) & = & \texttt{Tree}_{\texttt{abs}}\ (\texttt{In1}\ (\texttt{Leaf}\ (\texttt{Inr}\ a) \\
 & & \qquad \$\ \texttt{List}_{\texttt{REP}}\ (\texttt{List\_map}\ \texttt{Tree}_{\texttt{rep}}\ ts))
\end{array}
$$

0-ary constructors are represented using `Numb(0)`, i.e. the representation of the only element of the unit type.

7 Theorems

For a datatype T, the induction theorem expresses the wellknown principle of structural induction over type T. For its proof, note that the inductive characterization of the representing set gives us an induction theorem for that set. The induction theorem for T is obtained directly by lifting this theorem to T using $T_{\texttt{abs}}$. For our example type, it reads as follows:

$$
\frac{\begin{array}{l}\forall\ a.\ P\,(\texttt{LEAF a}) \\ \forall\, b.\ \forall\ ts.\ ts \in \texttt{List\_set}\ \{\, t.\ P\ t\,\} \implies P\,(\texttt{NODE}\,(b,\, ts))\end{array}}{\forall\ t.\ P\ t}
$$

Primitive recursive functions are the unique functions solving certain functional equalities. In our example, this means that for a pair of functions

$$
\begin{array}{lcl}
h_1 & : & 'a \Rightarrow 'c \\
h_2 & : & ['b, ('a, 'b)\,\texttt{Tree List}, 'c\ \texttt{List}] \Rightarrow 'c
\end{array}
$$

there exists a unique function f satisfying

$$
\begin{array}{lcl}
f\ (\texttt{LEAF}\,a) & = & h_1\,a \\
f\ (\texttt{NODE}\ (b,\ ts)) & = & h_2\,(b,\ ts,\ \mathit{List}\texttt{\_map}\ f\ ts)
\end{array}
$$

Theorems justifying primitive recursion can be proven by an inductive construction of the graph of the function, c.f. [3].

In [1], a different notion of primitive recursion is suggested. In the case of type `Tree` for example, it is required that given

$$\begin{array}{lcl} h_1 & : & 'a \Rightarrow 'c \\ h_2 & : & ['b,\ ('a, 'b)\ \texttt{Tree List},\ 'd] \Rightarrow 'c \\ h_3 & : & 'd \\ h_4 & : & [('a, 'b)\,\texttt{Tree},\ ('a, 'b)\,\texttt{Tree List},\ 'c,\ 'd] \Rightarrow 'd \end{array}$$

there exists a unique pair of functions f and g satisfying

$$\begin{array}{lcl} f\ (\texttt{LEAF}\ a) & = & h_1\ a \\ f\ (\texttt{NODE}\,(b,\ ts)) & = & h_2\,(b,\ ts,\ g\ ts) \\ g\ [\,] & = & h_3 \\ f\ (\texttt{Cons}\,(t,\ ts)) & = & h_4\,(t,\ ts, f\ t,\ g\ ts) \end{array}$$

For the type `Tree`, we have proven that this agrees with our notion of primitive recursive functions. It seems that this also holds for arbitrary datatypes, but we have not proven this formally yet.

8 Concluding remarks

We have presented an approach to the representation of datatypes which applies also to datatypes T with recursive occurrences of T inside of type expressions. Our approach is based on two generic functions T`_map` and T`_set` which can be defined for an arbitrary datatype T. The use of these functions has led us also to a different formulation of the induction theorem and of primitive recursive functions than usual.

The discussion of representations in the first four sections applies also to co-datatypes and to mutual recursive types. By suitable adjustments of the material in the last sections, an extension of our approach to these types should be possible.

Acknowledgements

I would like to thank M. Cieliebak for his comments.

References

[1] E. 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*, volume A-20 of *IFIP Transactions*, pages 561–568, Leuven, Belgium, 1992. North-Holland.

[2] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM*, 24:68–95, 1977.

[3] John Harrison. Inductive definitions: automation and application. Proceedings of the 1995 International Workshop on the HOL theorem proving system and its applications, Aspen Grove, Utah. 1995. To appear.

[4] E. Meijer, M.M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Functional Programming and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

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

[6] Larry Paulson. Co-induction and co-recursion in higher-order logic. Technical Report 304, Computer Laboratory, University of Cambridge, England, 1993.

[7] Philip Wadler. Monads for functional programming. In *Lecture notes for Marktoberdorf Summer School on Program Design Calculi, Springer-Verlag*, 1992.

[8] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. Foundations of Computing. The MIT Press, 1993.