Recursive Definitions in Z

R.D. Arthan

Lemma 1 Ltd.
2nd Floor,
31A Chain St.,
Reading,
Berkshire,
UK. RG10 9NX
rda@lemma-one.com

Abstract. This paper considers some issues in the theory and practice of defining functions over recursive data types in Z. Principles justifying such definitions are formulated. Z free types are contrasted with the free algebras of universal algebra: the notions turn out to be related but not isomorphic.

1 Introduction

The consistency of a Z specification is a matter of some practical importance. Effort expended in reasoning about an inconsistent specification is wasted and implementation of an inconsistent specification is either impossible or trivial depending on one's point of view. The most widely used definitions of Z, [8, 9] do consider the consistency of some Z paragraph forms, most notably the free type paragraph. This topic is further explored in [1, 7, 11].

In this paper, we consider approaches to proving the consistency of a particular class of axiomatic description, namely, axiomatic descriptions that define functions on a (typically recursive) free type. We consider principles allowing us to verify the consistency of such definitions. These principles are not themselves proposed for inclusion in the language definition, since, as we shall see, they are logical consequences of the usual axioms for a free type. Instead we present them to serve as rules of thumb for authors and readers of specifications and as guidelines for implementors of tools.

Z is founded on set theory. It is instructive to compare the Z approach to recursive definitions with what one might find in notations with different foundations. In this paper, we compare the Z approach with an approach based on universal algebra. It turns out that there is a broad overlap, but that there also non-trivial differences mainly arising from the extra expressiveness that set theory affords.

We make free use of the Z toolkit. Techniques for informal reasoning about the toolkit are discussed in several books, e.g., [13]; progress on one approach to automated proof for the Z toolkit is reported in [2].

The rest of this paper is structured as follows: Sect. 2 discusses the practical issues; Sect. 3 considers general principles for definition by recursion; Sect. 4 contrasts the theory for Z with concepts from universal algebra; finally, Sect. 5 gives some concluding remarks.

2 Recursive Definitions in Practice

The following free type definition will provide a running example throughout this section. $\mathsf{BINTREE}$ comprises binary trees with integer labels at each node and leaf. The definition may readily be seen to be consistent using the methods of [1, 9]. Some example members of $\mathsf{BINTREE}$ are shown in Fig. 1.

$$\mathsf{BINTREE} ::= \mathsf{Leaf}\langle\!\langle \mathbb{Z} \rangle\!\rangle \mid \mathsf{Node}\langle\!\langle \mathbb{Z} \times \mathsf{BINTREE} \times \mathsf{BINTREE} \rangle\!\rangle$$

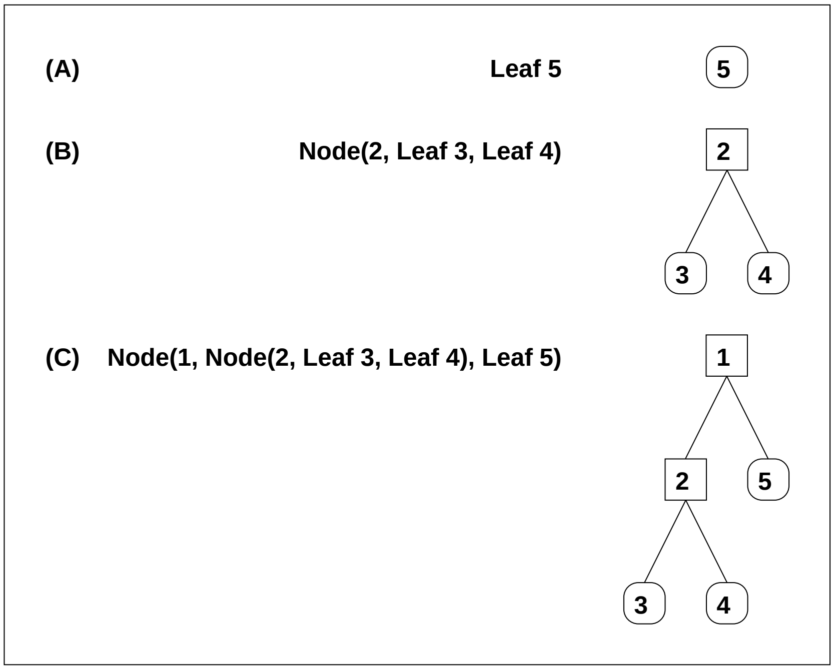

Fig. 1. Some Members of $\mathsf{BINTREE}$

2.1 Definition by Cases

We will be concerned with the consistency of functions defined using axiomatic descriptions. The functions will have for their domain our sample free type

$\mathsf{BINTREE}$. Our notion of consistency is straightforward: an axiomatic description defining a function f is consistent if we can prove, without using the axiomatic description, that there exists a function g satisfying the same constraints that the axiomatic description places on f. We will sometimes call such a g a witness to the consistency of f. Approaches to verifying the consistency of Z specifications are discussed in more detail in earlier work of the present author [1] and in a forthcoming paper by Sam Valentine [12]

As a first example of a function defined on the set $\mathsf{BINTREE}$, consider the following axiomatic description of a function intended to return the label at the root of a tree.

$$
\begin{array}{|l}
\mathsf{labelOf} : \mathsf{BINTREE} \to \mathbb{Z} \\
\hline
\forall\, i : \mathbb{Z} \bullet \mathsf{labelOf}(\mathsf{Leaf}\, i) = i \\
\forall\, i : \mathbb{Z};\ t_1, t_2 : \mathsf{BINTREE} \bullet \mathsf{labelOf}(\mathsf{Node}(i, t_1, t_2)) = i
\end{array}
$$

We may ask whether this axiomatic description is consistent, i.e., whether a function enjoying the properties we require of $\mathsf{labelOf}$ actually exists. Well, by the usual axioms that characterise a free type[1], any tree t in $\mathsf{BINTREE}$ is either $\mathsf{Leaf}\, i$ for some i or $\mathsf{Node}(i, t_1, t_2)$ for some i, t_1 and t_2. Moreover, Leaf and Node are injections and their ranges are disjoint, so that there is exactly one i for which t has one or other of these forms. Taking $\mathsf{labelOf}\, t$ to be this i gives the desired function. More formally, with each of the constructor functions, we may associate a corresponding destructor function defined by:

$$
\begin{aligned}
\mathsf{destLeaf} &== \mathsf{Leaf}^{-1} \\
\mathsf{destNode} &== \mathsf{Node}^{-1}
\end{aligned}
$$

We may then verify that these destructor functions are indeed functions and behave exactly as if they had been specified by the following axiomatic description:

$$
\begin{array}{|l}
\mathsf{destLeaf} : \mathrm{ran}\ \mathsf{Leaf} \to \mathbb{Z} \\
\mathsf{destNode} : \mathrm{ran}\ \mathsf{Node} \to (\mathbb{Z} \times \mathsf{BINTREE} \times \mathsf{BINTREE}) \\
\hline
\forall\, i : \mathbb{Z} \bullet \mathsf{destLeaf}(\mathsf{Leaf}\, i) = i \\
\forall\, i : \mathbb{Z};\ t_1, t_2 : \mathsf{BINTREE} \bullet \mathsf{destNode}(\mathsf{Node}(i, t_1, t_2)) = (i, t_1, t_2)
\end{array}
$$

A witness to the existence of our function $\mathsf{labelOf}$ can then be defined explicitly by the formula:

$$
\mathsf{destLeaf} \cup (\mathsf{destNode} \mathbin{⨾} (\lambda\, i : \mathbb{Z};\ t_1, t_2 : \mathsf{BINTREE} \bullet i))
$$

The use of destructor functions will enable us to justify most definitions of functions that pick apart the top level structure of an element of a free type. This pattern of definition may be called definition by cases; the principle of definition by cases, PDC, is a special case of the principle of definition by induction that we will now investigate.

---

[1] For a brief example and discussion of these axioms, see the proof of theorem 1 below. More leisurely expositions may be found in [1, 9].

2.2 Definition by Induction

The method of definition by cases is useful, but only gives us access to the top-level structure of an element of a free type. But we may need to dig deeper, e.g., to define a function to count the non-leaf nodes in a tree. Intuitively, the count is 0 for a leaf, and, for a non-leaf node, the count is 1 more than the sum of the counts for the children. This suggests the following axiomatic description of our function nodeCount:

$$
\begin{array}{|l}
\mathsf{nodeCount} : \mathsf{BINTREE} \to \mathbb{N} \\
\hline
\forall\, i : \mathbb{Z} \bullet \mathsf{nodeCount}(\mathsf{Leaf}\, i) = 0 \\
\forall\, i : \mathbb{Z};\ t_1, t_2 : \mathsf{BINTREE} \bullet \\
\quad \mathsf{nodeCount}(\mathsf{Node}(i, t_1, t_2)) = 1 + \mathsf{nodeCount}\, t_1 + \mathsf{nodeCount}\, t_2
\end{array}
$$

This definition is inductive, that is to say, the right-hand sides of the equations for nodeCount themselves involve uses of nodeCount. If we are given a concrete member of the free type, e.g., the tree C depicted in Fig. 1, then we can use the two equations in the axiomatic description as rewrite rules to evaluate nodeCount C:

$$
\begin{aligned}
\mathsf{nodeCount}\,\mathsf{C} &= \mathsf{nodeCount}(\mathsf{Node}(1, \mathsf{Node}(2, \mathsf{Leaf}\,3, \mathsf{Leaf}\,4), \mathsf{Leaf}\,5)) \\
&= 1 + \mathsf{nodeCount}(\mathsf{Node}(2, \mathsf{Leaf}\,3, \mathsf{Leaf}\,4) + \mathsf{nodeCount}(\mathsf{Leaf}\,5) \\
&= 1 + (1 + \mathsf{nodeCount}(\mathsf{Leaf}\,3) + \mathsf{nodeCount}(\mathsf{Leaf}\,4)) + 0 \\
&= 1 + (1 + 0 + 0) + 0 \\
&= 2
\end{aligned}
$$

However, any attempt to use the second equation as a rewrite rule will fail, in general, to eliminate nodeCount from an expression of the form nodeCount u if u involves variables: e.g., without knowing the value of x, we cannot simplify nodeCount x.

How are we to know that our axiomatic description of nodeCount is consistent? Appealing to a general theorem we shall discuss later, we can use the following principle of definition by induction (PDI) for the free type BINTREE:

$$
\begin{aligned}
[Y] \vdash\ & \forall\ e_1 : \mathbb{Z} \to Y;\ e_2 : (\mathbb{Z} \times Y \times Y) \to Y \bullet \\
& \quad \exists_1\ h : \mathsf{BINTREE} \to Y \bullet \\
& \quad\quad (\forall\, i : \mathbb{Z} \bullet h(\mathsf{Leaf}\, i) = e_1\, i) \\
& \quad \wedge\ (\forall\, i : \mathbb{Z};\ t_1, t_2 : \mathsf{BINTREE} \bullet h(\mathsf{Node}\,(i, t_1, t_2)) = e_2(i, h\, t_1, h\, t_2))
\end{aligned}
$$

The notation here means that the theorem PDI is generic in Y, i.e., we may instantiate Y to any set of any type. The theorem is concerned with the problem of defining a function $h : \mathsf{BINTREE} \to Y$ satisfying the given equations. The functions e_1 and e_2 are the data of the problem: e_1 specifies how h is to behave on leaves; e_2 specifies how the results of applying h to the children of a node are to be combined to give the value of h at that node. The theorem asserts that every such problem has a unique solution.

For our node-counting function, we use $Y = \mathbb{N}$ and take the data e_1 and e_2 to be:

$$\begin{aligned} e_1 &= (\lambda\, i : \mathbb{Z} \bullet 0) \\ e_2 &= (\lambda\, i : \mathbb{Z};\ t_1, t_2 : \mathbb{N} \bullet 1 + t_1 + t_2) \end{aligned}$$

Applying PDI, we can conclude that there exists a function $h : \mathsf{BINTREE} \to \mathbb{N}$ such that, for all integers, i, and trees, t_1 and t_2 the following equations hold:

$$\begin{aligned} h(\mathsf{Leaf}\, i) &= e_1\ i = 0 \\ h(\mathsf{Node}(i, t_1, t_2)) &= e_2(i, h\ t_1, h\ t_2) = 1 + h\ t_1 + h\ t_2 \end{aligned}$$

These are precisely our requirements for nodeCount so this h supplies a witness to the consistency of our definition of nodeCount. As an exercise, the reader may wish to find the data functions e_1 and e_2 that give the height of a tree:

$$\begin{array}{|l} \mathsf{height} : \mathsf{BINTREE} \to \mathbb{N} \\ \hline \forall\, i : \mathbb{Z} \bullet \mathsf{height}(\mathsf{Leaf}\, i) = 0 \\ \forall\, i : \mathbb{Z};\ t_1, t_2 : \mathsf{BINTREE} \bullet \\ \quad \mathsf{height}(\mathsf{Node}(i, t_1, t_2)) = 1 + max(\mathsf{height}\, t_1, \mathsf{height}\, t_2) \end{array}$$

2.3 Definition by Recursion

The principle of definition by induction, PDI is quite powerful, but has a limitation: in the function e_2, we can refer to the values of h on the children of a node, but we can't refer to the children themselves. As an example, assume we want a function which computes the sum over all non-leaf nodes in a tree of the node-heights weighted by the node-labels:

$$\begin{array}{|l} \mathsf{heightSum} : \mathsf{BINTREE} \to \mathbb{Z} \\ \hline \forall\, i : \mathbb{Z} \bullet \mathsf{heightSum}(\mathsf{Leaf}\, i) = 0 \\ \forall\, i : \mathbb{Z};\ t_1, t_2 : \mathsf{BINTREE} \bullet \\ \quad \mathsf{heightSum}(\mathsf{Node}(i, t_1, t_2)) = \\ \quad i * \mathsf{height}(\mathsf{Node}(i, t_1, t_2)) + \mathsf{heightSum}\, t_1 + \mathsf{heightSum}\, t_2 \end{array}$$

Here, since the function e_2 in PDI has no access to the values of the subtrees t_1 and t_2, PDI cannot be applied directly. Instead, we may appeal to the following principle of definition by recursion, PDR, which generalises PDI by making these subtrees available to the relevant data function in addition to the values of the recursive calls:

$$[Y] \vdash\ \forall\ d_1 : \mathbb{Z} \to Y;\ d_2 : \mathsf{BINTREE} \to (\mathbb{Z} \times Y \times Y) \to Y \bullet$$

$$
\begin{array}{l}
\exists_1\ h : \mathsf{BINTREE} \to Y \bullet \\
\quad (\forall\, i : \mathbb{Z} \bullet h(\mathsf{Leaf}\, i) = d_1\ i) \\
\land\ (\forall\, i : \mathbb{Z};\ t_1, t_2 : \mathsf{BINTREE} \bullet \\
\qquad h(\mathsf{Node}\,(i, t_1, t_2)) = d_2(\mathsf{Node}(i, t_1, t_2))(i, h\ t_1, h\ t_2))
\end{array}
$$

Note that PDI is, essentially, the special case of PDR in which the function d_2 makes no use of its first argument; thus PDR implies PDI. As we shall see in the proof of theorem 1 one can also derive PDR from PDI, so the two principles are equivalent.

In order to justify our definition of heightSum, let us use PDR, with $Y = \mathbb{Z}$, and with d_1 and d_2 given by:

$$
\begin{array}{l}
d_1 = (\lambda\, i : \mathbb{Z} \bullet 0) \\
d_2 = (\lambda\, b : \mathsf{BINTREE} \bullet \lambda\, i : \mathbb{Z};\ t_1, t_2 : \mathbb{N} \bullet i * \mathsf{height}\, b + t_1 + t_2)
\end{array}
$$

With this data, PDR delivers us a function h that is just what we need to justify the consistency of our axiomatic description of heightSum.

3 Theoretical Issues

3.1 PDR: The General Case

In Sect. 2, we have made several appeals to a general principle of definition by recursion (PDR). We now describe this principle in the general case. As the reader will see, the general principle is rather lengthy to describe.

So, let us consider a general free type definition. The free type definition has m nullary constructors, $\alpha_1, \ldots \alpha_m$ and n non-nullary constructors $\beta_1, \ldots \beta_n$. For simplicity, we assume that the nullary constructors are given first. Since the order of the branches in a free type is immaterial, this gives no real loss of generality. Our general free type definition thus has the form:

$$
\mathcal{T} ::= \alpha_1 \mid \ldots \mid \alpha_m \mid \beta_1 \langle\!\langle E_1 \rangle\!\rangle \mid \ldots \mid \beta_n \langle\!\langle E_n \rangle\!\rangle
$$

The type rules for Z require that each expression E_i be a set of elements of some type τ_i say. Here the type τ_i is some expression built up from ground types (and $\mathcal{T}$) using cartesian product, power sets and schema type constructions. Since the elements of each set E_i have type τ_i, the soundness of the Z type system ensures that $E_i \in \mathbb{P}\, \tau_i$.

For example, consider the following type of trees, with leaves either empty or labelled with a sequence of binary trees (these trees being represented using the free type BINTREE discussed in Sect. 2), and with branches constructed using a schema type:

$$
\mathsf{EG} ::= \mathsf{Empty} \mid \mathsf{TreeList}\langle\!\langle \mathrm{seq}\ \mathsf{BINTREE} \rangle\!\rangle \mid \mathsf{EGNode}\langle\!\langle [a, b : \mathsf{EG}] \rangle\!\rangle
$$

.

$\mathcal{T}$	EG
m	1
n	2
α_1	Empty
β_1	TreeList
E_1	seq BINTREE
β_2	EGNode
E_2	$[a, b : \mathsf{EG}]$
τ_1	$\mathbb{P}(\mathbb{Z} \times \mathsf{BINTREE})$
τ_2	$[a, b : \mathsf{EG}]$

Table 1. Metanotation for the Example

Remembering that sequences are partial functions on the integers and that functions are just sets of pairs, we see that in this example our metavariables, $\mathcal{T}$, m, n, etc., take the values shown in table 1.

Returning to the general case, PDR will be a theorem asserting the existence of solutions to problems of a certain kind. The theorem will have a generic parameter Y and will be constructed from the following components:

Data: the data comprise m elements of Y and n two-argument functions. The second argument of each function h_i ranges over the set, $\tau_i[Y/\mathcal{T}]$ obtained by substituting Y for each occurrence of $\mathcal{T}$ in the type τ_i:

$$\begin{aligned} c_1, \ldots, c_m &: Y \\ d_1 &: \mathcal{T} \to \tau_1[Y/\mathcal{T}] \to Y \\ &\vdots \\ d_n &: \mathcal{T} \to \tau_n[Y/\mathcal{T}] \to Y \end{aligned}$$

Solution: the solution is a function h:

$$h : \mathcal{T} \to Y$$

Condition: the condition comprises $m + n$ equations. The first m of these correspond to the data elements $c_1, \ldots c_m$ and are easy to state:

$$\begin{aligned} h\,\alpha_1 &= c_1 \\ &\vdots \\ h\,\alpha_m &= c_m \end{aligned}$$

To state the remaining n equations, we need to represent the notion of a "recursive call" of the function h. To do this we need to use the structure of the types $\tau_1, \ldots \tau_n$. We think of an element of the type τ as an expression tree formed using tuples, bindings and set constructions. Some leaves of this expression tree correspond to recursive appearances of $\mathcal{T}$ in τ and are labelled with elements of $\mathcal{T}$. Other leaves of the expression tree corresponding to ground types, G, say are labelled with elements of G. We need to describe the function from τ to $\tau[Y/\mathcal{T}]$ that works by replacing each leaf label $t \in \mathcal{T}$ by $h\ t$ and leaving other leaf labels as they are. The following function h_τ defined by induction over the structure of τ does the job:

$$h_{\mathcal{T}}\ x = h\ x \tag{1}$$
$$h_G\ y = y \tag{2}$$
$$h_{(\tau \times \ldots)}(x, \ldots) = (h_\tau\ x, \ldots) \tag{3}$$
$$h_{[a:\tau;\ \ldots]} \langle\!|\ a == x, \ldots, |\!\rangle = \langle\!|\ a == h_\tau\ x, \ldots, |\!\rangle \tag{4}$$
$$h_{(\mathbb{P}\,\tau)}\ A = h_\tau (\!|\ A\ |\!) \tag{5}$$

Here the five clauses correspond to: (1) $\mathcal{T}$ itself, (2) some other ground type G, (3) a cartesian product, (4) a schema type, and (5) a set type. The argument of h_τ in each clause represents a general element of τ, so the domain of h_τ is τ and, as ran $h \subseteq Y$, the range of h_τ is contained in $\tau[Y/\mathcal{T}]$. The notation $\langle\!|\ a == x, b == y, \ldots\ |\!\rangle$ denotes a binding with a component a with value x, a component b of value y and so on.

To sum up, the above construction provides functions h_τ for each Z type τ; h_τ acts on an element, t, of τ by mapping h over the recursive appearances of members of $\mathcal{T}$ inside t. If a branch of the free type is not recursive (so that $\mathcal{T}$ does not appear in τ), the corresponding h_τ will be $\mathsf{id}\,\tau$. For example, for the types τ_1 and τ_2 that arise in the definition of EG, the functions h_{τ_1} and h_{τ_2} are as follows[2]:

$$\begin{aligned}
h_{\tau_1} &= (\lambda\, A : \mathbb{P}(\mathbb{Z} \times \mathsf{BINTREE}) \bullet (\lambda\, y : \mathbb{Z} \times \mathsf{BINTREE} \bullet y)(\!|\ A\ |\!)) \\
&= \mathsf{id}\ \mathbb{P}(\mathbb{Z} \times \mathsf{BINTREE}) \\
h_{\tau_2} &= (\lambda\, t : [a, b : \mathsf{EG}] \bullet \langle\!|\ a == h\,(t.a), t_2 == h\,(t.b)\ |\!\rangle)
\end{aligned}$$

Armed with h_τ, we can now give the remaining n equations to complete the condition of the PDR problem:

$$\begin{aligned}
h(\beta_1\ e_1) &= d_1\ (\beta_1\ e_1)\,(h_{\tau_1}\ e_1) \\
&\vdots \\
h(\beta_n\ e_n) &= d_n\ (\beta_1\ e_n)\,(h_{\tau_n}\ e_n)
\end{aligned}$$

[2] We have been slightly lax in giving equations to define the functions h_τ without making their domains explicit; a tool that automated the theory could either use λ-expressions for the functions, as we have done in this example, or introduce axiomatic descriptions for the functions that map h to h_τ and use those to abbreviate the predicates.

To give the formal statement of PDR for the general case, we combine the above pieces in the form $\forall$ *data* $\bullet\ \exists_1$ *solution* $\bullet$ *conditions*:

$$
\begin{array}{l}
[Y] \vdash \forall\ \ c_1, \ldots, c_m : Y; \\
\qquad\qquad\qquad d_1 : \mathcal{T} \rightarrow \tau_1[Y/\mathcal{T}] \rightarrow Y; \\
\qquad\qquad\qquad\quad \vdots \\
\qquad\qquad\qquad d_n : \mathcal{T} \rightarrow \tau_n[Y/\mathcal{T}] \rightarrow Y \bullet \\
\quad \exists_1\ h : \mathcal{T} \rightarrow Y \bullet\ h\,\alpha_1 = c_1 \wedge \\
\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad\quad h\,\alpha_m = c_m \wedge \\
\qquad\qquad\qquad\quad (\forall\, e_1 : E_1 \bullet h(\beta_1\ e_1) = d_1\ (\beta_1\ e_1)\ (h_{\tau_1}\ e_1)) \wedge \\
\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad\quad (\forall\, e_n : E_n \bullet h(\beta_n\ e_n) = d_n\ (\beta_1\ e_n)\ (h_{\tau_n}\ e_n))
\end{array}
$$

PDI is the special case of the above assertion PDR in which the functions d_i make no use of their first argument, in which case, we can reformulate the assertion to remove the first argument. Similarly, PDC is essentially the special case of PDR in which the d_i make no use of their second argument.

For example, using the relevant values for h_τ computed above, PDR for the free type EG is the following assertion:

$$
\begin{array}{l}
[Y] \vdash \forall\ \ c_1 : Y; \\
\qquad\quad d_1 : \mathsf{EG} \rightarrow \mathbb{P}(\mathbb{Z} \times \mathsf{BINTREE}) \rightarrow Y; \\
\qquad\quad d_2 : \mathsf{EG} \rightarrow [a, b : Y] \rightarrow Y \\
\quad \exists_1\ h : \mathsf{EG} \rightarrow Y \bullet \\
\qquad h\ \mathsf{Empty} = c_1 \wedge \\
\qquad (\forall\, e_1 : \mathrm{seq}\ \mathsf{BINTREE} \bullet h(\mathsf{TreeList}\ e_1) = d_1\ (\mathsf{TreeList}\ e_1)\ e_1) \wedge \\
\qquad (\forall\, e_2 : [a, b : \mathsf{EG}] \bullet h(\mathsf{EGNode}\ e_2) = \\
\qquad\qquad d_2\ (\mathsf{EGNode}\ e_2)\ \langle\!|\ a == h\,(e_2.a), b == h\,(e_2.b)\ |\!\rangle)
\end{array}
$$

At the price of complicating the description, our formulation of PDR is clearly amenable to some improvement. For example, the data functions for non-recursive branches (such as d_1 in the above example) have two arguments which contain the same information in the statement of the theorem; the extra argument can be removed. If the i-th branch of the free type is recursive, then the first argument of the function d_i is always $\beta\ e_i$, and arguably it would be better to use e_i instead. However, the latter proposal will usually make the declaration of d_i longer.

A further improvement that might be sugggested would be to use $E_i[Y/\mathcal{T}]$ rather than $\tau_i[Y/\mathcal{T}]$ for the second domain of the data functions d_i. Typically, $E_i[Y/\mathcal{T}]$ will be a simpler expression than $\tau_i[Y/\mathcal{T}]$, as happens for d_1 in our example: the declaration would be shorter and clearer if we could use seq BINTREE in place of $\mathbb{P}(\mathbb{Z} \times \mathsf{BINTREE})$. However, as we shall see in section 4, this suggestion does not work in general.

3.2 Proof of PDR and PDI

In this section, we present the main theoretical result of this document which states that the principle of definition by recursion (PDR) is a consequence of the usual axioms that characterise a free type as described in [1, 9]. Thus one can safely make free use of PDR once a free type definition is known to be consistent.

Theorem 1. PDR *and* PDI *are consequences of the usual axioms that characterise a free type.*

Proof: while no really creative work is required, the details of the proof are not entirely trivial. To state and prove the general result, it would probably be best to use the framework used in [1] to simplify the syntactic complications. For present purposes, we will proceed by example and just demonstrate the result for a particular free type. It is convenient to consider a free type with just one constructor, and so we will use the following free type FT representing trees with arbitrary finite unordered branching. Some elements of FT are shown in figure 2.

$$\mathsf{FT} ::= \mathsf{k}\langle\!\langle \mathbb{F}\,\mathsf{FT} \rangle\!\rangle$$

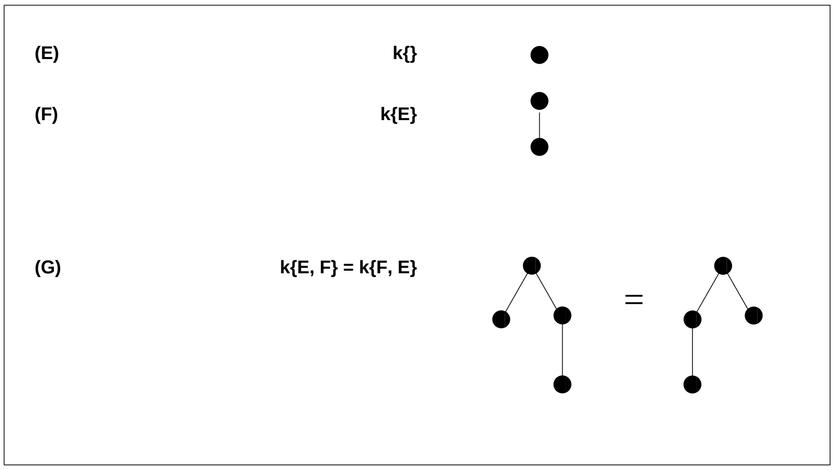

Fig. 2. Some Members of FT

The "usual axioms" for this free type amount to the following assertions: *(i)* the constructor function k is injective and surjective; and *(ii)* the following principle of proof by induction (PPI) holds:

$$\forall\, W : \mathbb{P}\,\mathsf{FT} \bullet \mathsf{k}(\!|\, \mathbb{F}\, W \,|\!) \subseteq W \Rightarrow W = \mathsf{FT}$$

That is to say the only subset W of FT that is closed under formation of new trees from old is FT itself. This principle allows us to reason by structural induction over trees: to show that a property $P(x)$ holds for every $x \in \mathsf{FT}$, it suffices to show that if $A \in \mathbb{F}\,\mathsf{FT}$ is such that $P(y)$ holds for every $y \in A$, then also $P(\mathsf{k}(A))$ holds.

To prove the theorem, it is slightly easier and somewhat more informative to prove PDI first and then derive PDR from that. To demonstrate PDI for the free type FT, we must prove the following assertion (with generic parameter Y):

$$[Y] \vdash \forall\, e : \mathbb{P}\, Y \to Y \bullet \exists_1\, h : \mathsf{FT} \to Y \bullet \forall\, x : \mathbb{F}\,\mathsf{FT} \bullet h(\mathsf{k}(x)) = e(h(\!|\; x \;|\!)) \qquad (6)$$

The proof of (6) is similar to the proof of definition by induction for the natural numbers that one can find in elementary texts on set theory (e.g., [5]). Given Y and e as in the statement of the theorem, we consider partial approximations to the desired total function h, That is to say we consider functions $g : \mathsf{FT} \to Y$ which satisfy $g(\mathsf{k}(x)) = e(g(\!|\; x \;|\!))$ whenever both sides of that equation are defined. We show that any two such approximations g_1 and g_2 are compatible, i.e., their union is again a function. We then find that the union of all such approximations g turns out to be the desired total function h.

More formally, let us define a family J of subsets of FT as follows:

$$\begin{aligned} J = \{A : \mathbb{P}\,\mathsf{FT} \mid \exists\, g : A \to Y \bullet \forall\, x : \mathbb{F}\,\mathsf{FT} \bullet \\ \mathsf{k}(x) \in A \Rightarrow x \subseteq A \wedge g(\mathsf{k}(x)) = e(g(\!|\; x \;|\!))\} \end{aligned}$$

We now make two claims about J:

Claim (A): Let $A_1, A_2 \in J$, and let $g_i : A_i \to Y$ be the functions whose existence is asserted by the definition of J $(i = 1, 2)$, then $(A_1 \cap A_2) \lhd g_1 = (A_1 \cap A_2) \lhd g_2$. That is to say, the functions g_1 and g_2 are *compatible*: their union $g_1 \cup g_2$ is also a function (from $A_1 \cup A_2$ to Y).

Claim (B): Let $A \in J$, and $x \in \mathbb{F}\, A$, then $(A \cup \{\mathsf{k}(x)\}) \in J$. More precisely, if the set A is not closed under the constructor k, so that, for some finite subset x of A, $\mathsf{k}(x) \notin A$, then we can extend a partial approximation $g : A \to Y$ to give a partial approximation $g' : (A \cup \{\mathsf{k}(x)\}) \to Y$.

We will only sketch the proofs of these claims: *ad (A)*, one defines a set W by $W = \{y : \mathsf{FT} \mid y \in A_1 \cap A_2 \Rightarrow g_1(y) = g_2(y)\}$, i.e., W is the set of points at which g_1 and g_2 agree, when they are both defined, and one then shows using PPI that $W = \mathsf{FT}$; *ad (B)*, one defines the extension g' of g to agree with g on A and to take the value $e(g(\!|\; x \;|\!))$ on $\mathsf{k}(x)$, and one then checks (using the injectivity of k) that this does indeed define a function on $A \cup \{\mathsf{k}(x)\}$ with the necessary properties.

Taking $A_1 = A_2$ in claim *(A)*, we see that any two approximations g_1 and g_2 with common domain $A = A_1 = A_2$ are identical. In particular, if $A = \mathsf{FT}$, the uniqueness part of (6) follows and it is only the existence of the function h that we have left to prove.

Let us now define $Q \subseteq \mathsf{FT}$, and a function $q : Q \to Y$, by:

$$Q = \bigcup J$$

$$q = \bigcup\{g : \mathsf{FT} \nrightarrow Y \mid \exists A : J \bullet g \in A \to Y \wedge \\ \forall x : \mathbb{F}\,\mathsf{FT} \bullet \mathsf{k}(x) \in A \Rightarrow x \in \mathbb{F}\,A \wedge h(\mathsf{k}(x)) = e(h(\!|\ x\ |\!)) \}$$

That q is indeed a function follows from claim *(A)*, which says that the functions whose union is formed in the definition of q are compatible. Note also that by the definitions of J, Q, and q, Q is indeed the domain of the function q.

We now make two further claims:

Claim(C) $Q \in J$.

Claim(D) $Q = \mathsf{FT}$

Again we will only sketch the proofs: *ad (C)*, one checks that q will serve as a partial approximation to h on Q and so concludes that Q belongs to J; *ad (D)*, one uses claims *(B)* and *(C)* to show that Q is closed under the constructor k and concludes from PPI that $Q = \mathsf{FT}$.

To complete the proof of (6), we deduce from claims *(C)* and *(D)* that $\mathsf{FT} \in J$; then the definition of J furnishes us with a function $g : \mathsf{FT} \to Y$ satisfying:

$$\forall x : \mathbb{F}\,\mathsf{FT} \bullet \mathsf{k}(x) \in \mathsf{FT} \Rightarrow x \in \mathbb{F}\,\mathsf{FT} \wedge g(\mathsf{k}(x)) = e(g(\!|\ x\ |\!)) \tag{7}$$

(In fact, $g = q$, but we no longer need the details of our explicit construction of q.) As $\mathsf{k} \in \mathbb{F}\,\mathsf{FT} \to \mathsf{FT}$, $\mathsf{k}(x) \in \mathsf{FT}$ for any $x \in \mathbb{F}\,\mathsf{FT}$; so, taking $h = g$, (7) simplifies to:

$$\forall x : \mathbb{F}\,\mathsf{FT} \bullet h(\mathsf{k}(x)) = e(h(\!|\ x\ |\!))$$

Thus this choice of h has the properties we need to complete the proof of (6).

To derive PDR from PDI, what we have to do is prove the following, using (6) as an assumption:

$$[Y] \vdash \forall d : \mathsf{FT} \to \mathbb{P}\,Y \to Y \bullet \exists_1 h : \mathsf{FT} \to Y \bullet \\ \forall x : \mathbb{F}\,\mathsf{FT} \bullet h(\mathsf{k}\,x) = d(\mathsf{k}\,x)(h(\!|\ x\ |\!)) \tag{8}$$

So let us assume that (6) holds (as indeed it does, since we have just proved it). Assume that Y and d as above are given. We have to use (6) to construct a function h satisfying the above equation.

In the general case, one must account for the possibility that the free type defines an empty set (in which case both PDI and PDR may be seen to hold vacuously); for FT, we know that the free type is non-empty, and using (6) one can deduce that Y is also non-empty (otherwise the function d could not exist).

Given that FT and Y are not empty, let us choose elements $r_0 \in \mathsf{FT}$ and $y_0 \in Y$. Using these elements we define a function $e \in \mathbb{P}(\mathsf{FT} \times Y) \to \mathsf{FT} \times Y$ to which we can apply (6) as follows:

$$e(s) = \begin{cases} \mathsf{k}(\mathsf{first}(\!|\ s\ |\!)), d(\mathsf{k}(\mathsf{first}(\!|\ s\ |\!)))(\mathsf{second}(\!|\ s\ |\!)) & \text{if } \mathsf{first}(\!|\ s\ |\!) \text{ is finite} \\ (r_0, y_0) & \text{otherwise} \end{cases}$$

(Here the value of e is only relevant on sets s for which $\mathsf{first}(\!|\ s\ |\!)$ is finite; however, to apply (6) in the form in which we have stated it, we need e to be total on $\mathbb{P}(\mathsf{FT} \times Y)$, and so we have used r_0 and y_0 to extend the relevant parts of e to give a total function delivering an arbitrary fixed result on the irrelevant values of s.)

By (6), we know that there is a unique function $g \in \mathsf{FT} \to \mathsf{FT} \times Y$ such that:

$$\forall x : \mathbb{F}\,\mathsf{FT} \bullet g(k(x)) = e(g(\!|\ x\ |\!))$$

We then define the desired function h by:

$$h = \mathsf{second} \circ g$$

One may now check using the various definitions that h satisfies the following equation, which completes the proof of the existence part of (8).

$$\forall x : \mathbb{F}\,\mathsf{FT} \bullet h(\mathsf{k}(x)) = d(\mathsf{k}\,x)(h(\!|\ x\ |\!))$$

The uniqueness part of (8) follows (with a little extra work) from the uniqueness of g. This completes the proof of our theorem in the special case of the free type FT. The proof in the general case is very similar in structure: using the terminology of [1], a monotonic operator ϕ would appear where the proof for FT has the finite set operator, $\mathbb{F}$; the type τ of this operator would appear in place of $\mathbb{P}$ where appropriate; and the use of relational image to map the function being defined over subtrees would be replaced by use of the operator $\hat{\tau}$ (cf. h_τ in the notation of section 3.1 above). The derivation of PDR from PDI in the general case involves some reasoning about the functorial properties of $\hat{\tau}$; to bypass this, it is not difficult to generalise the argument given above for PDI to deliver PDR directly. $\square$

4 Comparison with Universal Algebra

Many people may have wondered why free types are called "free types". In this section, we shall explore the analogy that gives rise to the name, and find, perhaps surprisingly, that it is not as close as one might hope.

We will need some elementary ideas from universal algebra [4]. Universal algebra studies the features of algebraic systems such as rings, groups, fields, etc., that are independent of the fine details of the theory of rings, groups, fields, etc. Two basic concepts in universal algebra are signatures and structures:

A *signature* is a syntactic construct defining some typed operators, for example, the signature corresponding to the theory of groups might have a nullary operator, $e : G$, a unary operator $\_^{-1} : G \to G$ and a binary operator $\_ \bullet \_ :$ $G \times G \to G$ giving the identity element, inverse operation and multiplication operation respectively.

A *structure* for a signature is a set provided with operations that implement the operators of the signature. The group of integers, for example, provides a structure for the signature described above taking $e = 0$, $x^{-1} = -x$ and $x \bullet y = x + y$.

We think of a signature as defining the set of all its structures. As an example, the following signature defines the set of all sets, X, equipped with a zero-element, Z and a function S from X to itself:

$$Z : X$$
$$S : X \to X$$

In describing structures for a given signature, we will use subscripts to distinguish different structures. Following this convention, an example of a structure for this signature might be to take $X_1 = \mathbb{N}$, $Z_1 = 0$ and $S_1 = succ$. Another might be to take $X_2 = \mathbb{Z}$, $Z_2 = 1$ and $S_2 = (\lambda\, i : \mathbb{Z} \bullet 2 * i)$. A function between two structures for the same signature is said to be a *morphism* if it commutes with the operations. For example, a function $f : X_1 \to X_2$ will be a morphism iff. $f\,0 = 1$ and $f(S_1\ i) = S_2(f\ i)$ for all i in X_1. One such morphism is the function that maps $i \in \mathbb{N}$ to $2^i \in \mathbb{Z}$, as one may readily check.

An isomorphism is a morphism that is also a bijection, and for many purposes isomorphic structures may be considered to be identical. Our example morphism from (X_1, Z_1, S_1) to (X_2, Z_2, S_2) is not a morphism, since it is not a surjection. However, if we take X_3 to be the set of non-negative powers of 2, take $Z_3 = 1$ and take $S_3 = (\lambda\, i : X_3 \bullet 2 * i)$, then mapping i to 2^i provides an isomorphism between (X_1, Z_1, S_1) and (X_3, Z_3, S_3) (which provides a discrete analogue of the principle that underlies slide-rules and log tables).

Subject to some restrictions on the types used in a signature, it turns out that any signature defines a non-empty set of structures and that the set of structures contains a distinguished class of structures called the *free algebras* for the signature. The characteristic property of a free algebra, A, is that for any other structure for the same signature, B, there exists a unique morphism from A to B. Any two free algebras for the same signature are isomorphic, and so we generally talk about "the" free algebra for a signature. For example the structure (X_1, Z_1, S_1) described above is the free algebra for its signature: this fact is essentially equivalent to a principle of definition by induction over the natural numbers (cf. the example morphisms given above).

We are concerned with the principles for definition by induction and recursion for a Z free type. The defining equations in these principles as exemplified in section 2 above are reminiscent of the equations that define a morphism in universal algebra. For example, PDI for the set of finite tree we used to illustrate the proof of theorem 1 contains the following equation:

$$h(\mathsf{k}(x)) = e(h(\!|\ x\ |\!))$$

If k and e here were operators for two different structures for a suitable signature, the above equation would say that the function h is a morphism between the two structures. In this light, we might hope that PDI would correspond to the defining property of a free algebra.

So, given a Z free type definition, we can construct a corresponding signature whose operators are the constructors of the free type. Each non-nullary operator is assigned the type of a function from the domain of the corresponding constructor to the unknown X. For example, the following free type definition corresponds to the signature with operators Z and S mentioned above:

$$\mathsf{Nat} ::= \mathsf{Z} \mid \mathsf{S}\langle\!\langle \mathsf{Nat} \rangle\!\rangle$$

In the sequel, given a Z free type definition such as the above, we can conveniently borrow the terminology of universal algebra and say that a structure for the free type is a tuple $(R, \mathsf{Z}_0, \mathsf{S}_0)$, where R is a set, $\mathsf{Z}_0 \in R$ and $\mathsf{S}_0 \in R \to R$.

For simple examples like the above, the analogy works quite well. However, the fact that one can write an arbitrary set-valued expression in a branch of a Z free type means that one can impose structural constraints[3]. that are not compatible with the usual methods of universal algebra. In the rest of this section, we will explore the somewhat unfortunate consequences of this aspect of free type definitions.

As a first observation on the technical problems that arise, we observe that the version of PDI or PDR that the theory of universal algebra delivers does not work so well for the signatures that can arise from Z free types. This relates to the question raised in the previous section of whether to use $\tau_i[Y/\mathcal{T}]$ or $E_i[Y/\mathcal{T}]$ in the statement of PDR or PDI. Consider for example the following type of unbalanced trees:

$$\mathsf{UBT} ::= \mathsf{Leaf}\langle\!\langle \mathbb{Z} \rangle\!\rangle \mid \mathsf{Node}\langle\!\langle \{i : \mathbb{Z};\ t_1, t_2 : \mathsf{UBT} \mid t_1 \neq t_2\} \rangle\!\rangle$$

For example, the trees depicted in Fig. 1 may all be viewed as members of UBT. The definition of UBT is consistent and the version of PDI that universal algebra gives us would be the following theorem:

$$\begin{aligned}
[Y] \vdash \forall\ & d_1 : \mathbb{Z} \to Y; \\
& d_2 : \{i : \mathbb{Z};\ t_1, t_2 : Y \mid t_1 \neq t_2\} \to Y \bullet \\
& \exists_1\ h : \mathsf{UBT} \to Y \bullet (\forall e_1 : \mathbb{Z} \bullet h(\mathsf{Leaf}\ e_1) = d_1\ e_1) \\
& \wedge\ (\forall e_2 : \{i : \mathbb{Z};\ t_1, t_2 : \mathsf{UBT} \mid t_1 \neq t_2\} \bullet \\
& \qquad h(\mathsf{Node}\ e_2) = d_2\ (e_2.1, h(e_2.2), h(e_2.3)))
\end{aligned}$$

In the notation of the previous section the above theorem uses $E_2[Y/\mathsf{UBT}]$ rather than $\tau_2[Y/\mathsf{UBT}]$ in the declaration of d_2; however, this is not strong enough to justify definitions that we might very well wish to make. For example, consider a function to count the non-leaf nodes in an unbalanced tree. This function, nodeCount say, ought to be defined just like the function of the same name in Sect. 2.2 using UBT instead of BINTREE. If we apply it to the tree B in Fig. 1, we will compute:

$$\begin{aligned}
\mathsf{nodeCount}\,\mathsf{B} &= 1 + \mathsf{nodeCount}(\mathsf{Leaf}\,3) + \mathsf{nodeCount}(\mathsf{Leaf}\,4) \\
&= 1 + (1 + 1) \\
&= 3
\end{aligned}$$

[3] We do not have complete freedom however; in particular, the type and scope rules of Z prevent us from referring to the constructors of the free type inside the set-valued expressions that appear on the right-hand side of the definition.

However, the data function d_2 required to carry out this computation is not a member of the set $\{i : \mathbb{Z};\ t_1, t_2 : \mathbb{Z} \mid t_1 \neq t_2\} \to \mathbb{Z}$, because $1 = 1$.

We can draw two conclusions from the above example. Firstly, the form of PDR using $\tau_i[Y/\mathcal{T}]$ as given in Sect. 3.1 is probably the one to use; secondly, we should be chary about drawing too much from the analogy of set-theoretic inductive definitions (i.e., Z free types) with universal algebra: the subjects are related but they are not isomorphic.

A deeper point at which the analogy breaks down highlights a potential misconception about the strength of PDR (or rather its equivalent PDI). In universal algebra terms, PDI should amount to the assertion that the type $\mathcal{T}$ in the statement of the theorem is the free algebra for its signature. Now, it is a fact of universal algebra that free algebras are unique up to isomorphism. The proof is elementary: if A and B are free algebras for the same signature, then there are unique morphisms $\alpha : A \to B$ and $\beta : B \to A$; now the composite $\alpha \mathbin{⨾} \beta$ is a morphism from A to itself, but so is $\mathsf{id}\, A$; so by the freeness of A, $\alpha \mathbin{⨾} \beta = \mathsf{id}\, A$, and similarly $\beta \mathbin{⨾} \alpha = \mathsf{id}\, B$; thus α and β are (mutually inverse) bijections and A and B are indeed isomorphic. It follows that PDI (or PDR) actually characterises the free algebra completely, and so, for example, other principles like the principle of proof by induction PPI[4] can be derived from it. Sadly, this reasoning does not transfer to Z free types, if we use our preferred formulation as in Sect. 3.1:

Theorem 2. *If we formulate* PDR *as in Sect. 3.1 (using $\tau_i[Y/\mathcal{T}]$ rather than $E_i[Y/\mathcal{T}]$), then* PDR *does not imply* PPI.

Proof: what we have to do is exhibit a set equipped with constructor functions for which PDR holds but PPI fails. To do this, let us consider the following free type definition:

$$\mathcal{D} ::= \mathsf{Z} \mid \mathsf{S}\langle\!\langle \{m : \mathcal{D} \mid \exists\, a, b : \mathcal{D} \bullet a \neq b\} \rangle\!\rangle$$

Here, the set expression $\{m : X \mid \exists\, a, b : X \bullet a \neq b\}$, is the empty set when X has less than 2 elements, and is equal to X otherwise. The free type definition $\mathcal{D}$ is therefore just like that for Nat in definition (9) above except that the "generative power" of the second branch is cut down to nothing when the free type is "small". PPI for a structure $(R, \mathsf{Z}_0, \mathsf{S}_0)$ for $\mathcal{D}$ is the following assertion:

$$\forall\, A : \mathbb{P}\, R \bullet (\mathsf{Z}_0 \in A \land \forall\, x : \{m : A \mid \exists\, a, b : A \bullet a \neq b\} \bullet \mathsf{S}_0(x) \in A) \Rightarrow A = R$$

The only possible structure for $\mathcal{D}$ with PPI must have the carrier set R equal to the singleton set $\{\mathsf{Z}_0\}$, because $\{m : A \mid \exists\, a, b : A \bullet a \neq b\}$ is empty if A is a singleton set.

Now, Nat has more than 1 element, so $\{m : \mathsf{Nat} \mid \exists\, a, b : \mathsf{Nat} \bullet a \neq b\} = \mathsf{Nat}$. It follows that Nat, Z and S give a structure for $\mathcal{D}$, and, moreover, PDR holds for

[4] See examples in the proof of theorems 1 and 2.

this structure, since the formal statement of PDR for this structure is equivalent to that for Nat. That is to say, the proposition:

$$\begin{array}{l}\forall\, c_1 : Y;\ d_0 : \{m : \mathsf{Nat} \mid \exists\, a, b : \mathsf{Nat} \bullet a \neq b\} \times Y \to Y \bullet \\ \quad \exists_1\, h : \{m : \mathsf{Nat} \mid \exists\, a, b : \mathsf{Nat} \bullet a \neq b\} \to Y \bullet \\ \quad\quad h(\mathsf{Z}) = c_1 \wedge \\ \quad\quad (\forall\, x : \{m : \mathsf{Nat} \mid \exists\, a, b : \mathsf{Nat} \bullet a \neq b\} \bullet h(\mathsf{S}(x)) = d_0(x, h(x)))\end{array}$$

is equivalent to:

$$\begin{array}{l}\forall\, c_1 : Y;\ d_0 : \mathsf{Nat} \times Y \to Y \bullet \\ \quad \exists_1\, h : \mathsf{Nat} \to Y \bullet \\ \quad\quad h(\mathsf{Z}) = c_1 \wedge (\forall\, x : \mathsf{Nat} \bullet h(\mathsf{S}(x)) = d_0(x, h(x)))\end{array}$$

which is the formal statement of PDR for Nat. Thus, we have a structure for $\mathcal{D}$ which satisfies PDR but not PPI, since, as we have already remarked, a structure satisfying PPI has to have the carrier set equal to a singleton set. □

In the proof of Theorem 2, the example involves a free type definition for which one of the branches is empty. It might be thought that this makes the example rather a special case which could be eliminated by an appropriate condition. However, there are more complex examples with PDR and not PPI which have no empty branches. Using an informal notation[5], one such is given by:

$$\mathcal{U} ::= u\langle\!\langle \{r : \mathbb{N} \leftrightarrow \mathcal{U} \mid \operatorname{ran} r \text{ is finite} \wedge (\mathcal{U} \text{ is countable} \Rightarrow r \in \operatorname{seq} \mathcal{U})\} \rangle\!\rangle$$

Here the least fixed point is the same as the countable set which is the least fixed point for the free type definition:

$$\mathcal{W} ::= w\langle\!\langle \operatorname{seq} \mathcal{W} \rangle\!\rangle$$

However, a fixed point for $\mathcal{U}$ with PDR but not PPI is given by the uncountable set which is the least fixed point for the following free type definition (cf. Example 6 in Sect. 2 of [1]):

$$\mathcal{Q} ::= q\langle\!\langle \{r : \mathbb{N} \leftrightarrow \mathcal{Q} \mid \operatorname{ran} r \text{ is finite}\} \rangle\!\rangle$$

In the terminology of [1], the idea behind these examples is as follows. We start with some monotonic operator, ϕ say with a least fixed point satisfying PDR. We want to use ϕ to construct an example with PDR but not PPI, and to give some room for manoeuvre, we arrange for the carrier set R to be a "large" set in some sense to be defined. We then construct another monotonic operator ψ, such that $\psi(X) \subseteq (\phi(X))$, for all X. We do this in such a way that if X is a "small" subset of R, then $\psi(X)$ is a lot smaller than $\phi(X)$, whereas if X is a "large" subset of R, $\psi(X) = \phi(X)$. It then turns out that a least fixed point for ϕ

[5] Formally, "X is finite" may be expressed in Z as "$X \in \mathbb{F}\, X$", and "X is countable" as "$\mathbb{N} \twoheadrightarrow X \neq \varnothing$".

with carrier set R say, may still have PDR with respect to ψ, since $\psi(R) = \phi(R)$, but R can contain "small" subsets which manage to be fixed points for ψ even though they are not fixed points for ϕ. In the case of $\mathcal{T}$ and $\mathcal{U}$ above "small" is interpreted as "no more than one element" and "countable" respectively.

Ultimately, the problem with formulating PDR as suggested by universal algebra, using $E_i[Y/\mathcal{T}]$, rather than $\tau_i[Y/\mathcal{T}]$ is that the resulting axiom involves application of the data functions d_i outside their domain of definition for bad cases like those we have just been looking at. Many of the free type definitions that arise in practice do not impose the structural constraints on the generative strengths of the constructors that give rise to this problem. In categorical language, these are the ones for which the operation mapping Y to $E_i[Y/\mathcal{T}]$ gives the objects part of a functor whose morphisms part is the operation h_τ discussed in Sect. 3.1 (see [3] for a more explicit account).

It may well be of benefit in developing conceptual and mechanized tools for working with Z free types to pay special attention to the functorial case[6] so as to exploit the additional properties it enjoys. This would let us use many of the practical techniques developed for other logical systems, e.g., see [6] for a treatment of recursive definitions in HOL. In particular, for functorial free type definitions, PDR actually entails all the other axioms and so may be used as a starting point for many lines of reasoning. It is perhaps unfortunate that this economy of axiomatisation is not available in the general case with which Z must deal.

5 Conclusions

The principles for defining functions over free types that we have articulated justify a useful class of definitions. The principles are intended to serve as rules of thumb for the ordinary Z practitioner and to guide those constructing tools supporting mechanized reasoning in Z.

The foundation of Z in set theory means that the theory of recursion for Z is perhaps less widely known in the formal methods world than might be the case for a notation founded in domain theory or universal algebra. While there is a broad overlap, the approaches are not isomorphic. For many purposes, the set-theoretic approach is, I would claim, both simpler and more powerful.

[6] A possible line in this direction would be to adopt the approach of [10] which reports on an enrichment of the Z type system, extending the basic types of Z to allow finer distinctions to be made by a Z support tool, e.g., to distinguish sequences from other less structured functions. If one required the richer types to be functorial, then the methods of [10] might enable a tool to recognise and exploit the functorial case in a systematic way. An alternative approach would be to use some heuristic method to derive from each expression E_i a more convenient expression to use in place of $\tau_i[Y/\mathcal{T}]$. Care would be needed in designing such a mechanism to ensure that the resulting definitional principle is sufficient for all purposes.

6 Acknowledgments

Much of the work reported in this paper was undertaken while the author was working for International Computers Ltd. under contract to the UK Government's Communications and Electronics Security Group. I am indebted to Colin Champion of CESG for permission to publish the results and to the ZUM'98 referees for their sympathetic and helpful comments.

References

1. R.D. Arthan. On Free Type Definitions in Z. In J.E. Nicholls, editor, *Z User Workshop, York 1991*. Springer-Verlag, 1992.
2. R.D. Arthan. Mechanizing the Z Toolkit. In Mike Mislova, editor, *Proceedings of the Oxford Workshop on Automated Formal Methods*. Elsevier Electronics Notes in Computer Science, 1997.
3. R.D. Arthan. Recursive Data Types in Typed Set Theory. *Unpublished pre-print*, 1997.
4. P.M. Cohn. *Universal Algebra*. D Reidel Publishing Company, 1981.
5. Paul R. Halmos. *Naive Set Theory*. Springer-Verlag, 1974.
6. Thomas F. Melham. Automating Recursive Type Definitions in Higher Order Logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*. Springer-Verlag, 1989.
7. A. Smith. *On Recursive Free Types in Z*. RSRE Memorandum 91028. MOD PE, RSRE, 1991.
8. J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 1989.
9. J.M. Spivey. *The Z Notation: A Reference Manual, Second Edition*. Prentice-Hall, 1992.
10. J.M. Spivey. Richer Types for Z. *Formal Aspects of Computing*, 8(5):565–584, 1996.
11. Michael Spivey. The Consistency Theorem for Free Type Definitions in Z (Short Communication). *Formal Aspects of Computing*, 8(3):369–376, 1996.
12. Sam Valentine. Inconsistency and Undefinedness in Z — A Practical Guide. *These proceedings*, 1998.
13. Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof*. Prentice/Hall International, 1996.