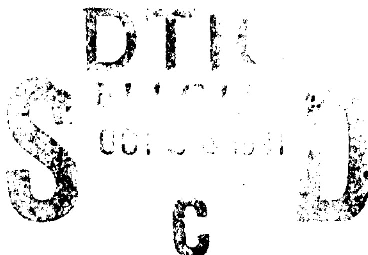
UNLIMITED

AD-A242 133

Report No. 91028

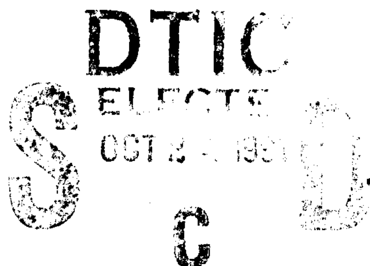
Report No. 91028

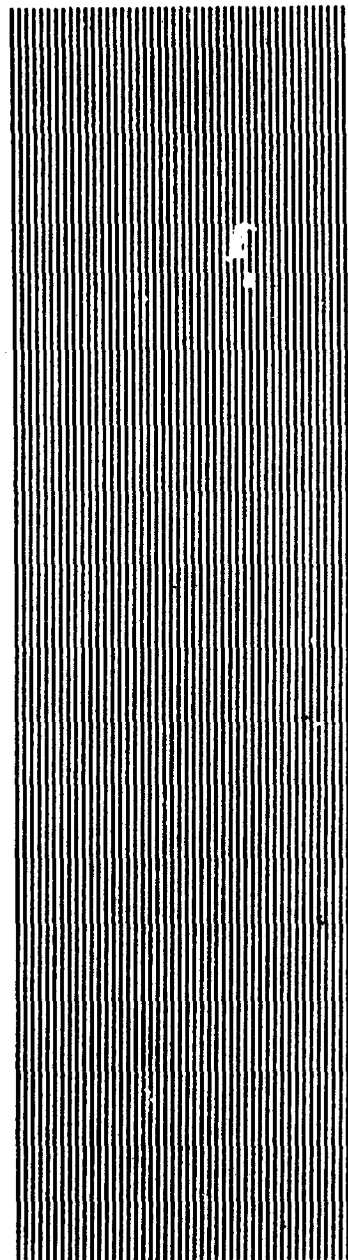
ROYAL SIGNALS AND RADAR ESTABLISHMENT, MALVERN



ON RECURSIVE FREE TYPES IN Z

Author: A Smith





91-13895

PROCUREMENT EXECUTIVE, MINISTRY OF DEFENCE
RSRE
Malvern, Worcestershire.

August 1991

91 10 23 028

UNLIMITED

CONDITIONS OF RELEASE

0109644 304797

** DRIC U

COPYRIGHT (c)
1988
CONTROLLER
HMSO LONDON

** DRIC Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Report 91028

Title: On Recursive Free Types in Z

Author: A. Smith

Date: August 1991

ABSTRACT

Inconsistent specifications may give rise to false conclusions in reasoning, thus destroying the point of having a specification. This report is concerned with inconsistent specifications which may arise when using the formal specification language Z. In particular, the report is concerned with the inconsistencies that can arise when using recursive free types, and recursive functions defined over recursive free types. The intended audience of the report consists of Z practitioners who wish to avoid writing meaningless specifications.

Copyright
©
Controller HMSO London
1991

Accession For
NTIS GRA&I
DTIC Tab
Unannounced
Justification
By
Distribution/
Availability Codes
Dist Avail and/or special
A-1

1 Introduction

In ordinary mathematics, an equation can be written down which is syntactically correct, but for which no solution exists. For example, consider the equation $x = x + 1$ defined over the real numbers; there is no value of x which satisfies it. Similarly it is possible to specify objects using the formal specification language Z [3,4], which can not possibly exist. Such specifications are called *inconsistent* and can arise in a number of ways.

Example 1

The following Z specification of a function f, from integers to integers

$$
\begin{array}{|l}
f : \mathbb{Z} \rightarrow \mathbb{Z} \\
\hline
\forall x : \mathbb{Z} \mid x \leq 0 \bullet f x = x + 1 \quad (i) \\
\forall x : \mathbb{Z} \mid x \geq 0 \bullet f x = x + 2 \quad (ii)
\end{array}
$$

is inconsistent, because axiom (i) gives $f\ 0 = 1$, while axiom (ii) gives $f\ 0 = 2$. This contradicts the fact that f was declared as a function, that is, f must have a unique result when applied to an argument. Hence no such f exists. Furthermore, if $f\ 0 = 1$ and $f\ 0 = 2$ then $1 = 2$ can be deduced! From $1 = 2$ anything can be deduced, thus showing the danger of an inconsistent specification.

Δ

Note that all examples and proofs start with the word *Example* or *Proof* and end with the symbol Δ.

1.1 Free types

Another way in which inconsistencies can arise in Z specifications is in the use of free types. Unlike given sets, a free type has some structure. Strictly, in Z, there is a difference between a type and its underlying set, but from here on, a type and its underlying set are regarded as equivalent. Z has a very powerful method for introducing free types, but this power can lead to inconsistencies. The general form for a free type definition is

$$T ::= c1 \mid \ldots \mid cm \mid d1 \langle\langle E1[T] \rangle\rangle \mid \ldots \mid dn \langle\langle En[T] \rangle\rangle \qquad (1)$$

This defines a new type T to be the labelled disjoint union of $c1, \ldots, cm$, $E1[T], \ldots, En[T]$. The $E1[T], \ldots, En[T]$ are set valued expressions which may involve T; if any of them do involve T then T is a *recursive* free type. The elements of T are the ci and anything of the form $di\ x$ where x is an element of $Ei[T]$. Expression 1 need not have any arms ci or it need not have any arms $di \langle\langle Ei[T] \rangle\rangle$, but obviously it must have at least one arm.

If T is a recursive free type, it must have one or more "base elements"; elements to enable other more complicated elements to be constructed. The following example illustrates this concept.

Example 2

A particular example of a free type is

$$T ::= a \mid b \langle\langle D \rangle\rangle \mid c \langle\langle F \times T \rangle\rangle$$

where D and F are given sets. T is a recursive free type, and comparing it with (1) gives

$$c1 = a,\ d1 = b,\ d2 = c,\ E1[T] = D,\ E2[T] = F \times T$$

The "base elements" of T come from the first two arms. They are a and anything of the form $b\ d$ for some element d of D. These can then be used to build up more complicated elements using the third arm, for example

$$c\ (f1, b\ d)$$

for some element $f1$ of F. These more complicated elements can then themselves be used to build up even more complicated elements, for example

$$c\ (f2, c\ (f1, b\ d))$$

for some element $f2$ of F, and so on.

Δ

The general free type definition (1) is simply shorthand for the following Z

[T]

$$\begin{array}{|l}
c1, \ldots, cm : T \\
d1 : E1[T] \rightarrowtail T \qquad\qquad\qquad\qquad (2) \\
\ldots \\
dn : En[T] \rightarrowtail T \\
\hline
disjoint\ \langle \{c1\}, \ldots, \{cm\}, ran\ d1, \ldots, ran\ dn \rangle \qquad (i) \\
\\
\forall W : \mathbb{P}T \bullet \\
\quad \{c1, \ldots, cm\} \cup d1 (\!| E1[W] |\!) \cup \ldots \cup dn (\!| En[W] |\!) \subseteq W \qquad (ii) \\
\quad \Rightarrow T \subseteq W
\end{array}$$

The ci are declared as elements of T, while the di are declared as injective functions (known as the constructors of T) from $Ei[T]$ to T. Axiom (i) states that the elements of $E1[T] \dots En[T]$ are mapped onto different elements of T, which are in turn different from the elements $c1, \dots, cm$ of T. Axiom (ii) is known as the *induction principle* for the free type, and can be used to prove statements of the form $\forall t : T \bullet P(t)$, for some property P, by structural induction. The expression $Ei[W]$ is obtained from $Ei[T]$ by replacing every free occurrence of T in $Ei[T]$ by W. A consequence of the induction principle is that T contains only the elements $c1, \dots, cm$ and those that can be constructed using $d1, \dots, dn$. It contains no other elements than these. Incidentally, as W has type $\mathbb{P}T$ in axiom (ii), and so $W \subseteq T$, then the sub-predicate $T \subseteq W$ of axiom (ii) may be replaced with $T = W$ if desired.

Example 3

For the free type in example 2, the general form in (2) becomes

$$[T]$$

$$\begin{array}{|l}
a : T \\
b : D \rightarrowtail T \\
c : (F \times T) \rightarrowtail T \\
\hline
disjoint \langle \{a\}, ran\ b, ran\ c \rangle \\
\\
\forall W : \mathbb{P}T \bullet \\
\quad \{a\} \cup b(\!|D|\!) \cup c(\!|F \times W|\!) \subseteq W \\
\quad \Rightarrow T \subseteq W
\end{array}$$

Δ

For some recursive free types, the objects specified in (2) can not possibly exist. To see how easily this can happen, consider examples 4, 5 and 6.

Example 4

Consider a programming language whose values are either booleans or functions involving booleans. To give a semantics for this language, using the specification language Z, the following free type might be used to express the values of the language

$$Value ::= bool \langle\!\langle \{T, F\} \rangle\!\rangle \mid fun \langle\!\langle Value \rightarrow Value \rangle\!\rangle$$

It states that values are either booleans or functions from values to values. From (2), one of the declarations is

$$fun : (Value \rightarrow Value) \rightarrowtail Value$$

But no such *fun* can exist, because for any set *Value*, the size of the set $Value \rightarrow Value$ is always greater than the size of *Value*. Thus there is no total injective function from $Value \rightarrow Value$ to *Value*. So the free type *Value* can not possibly exist, and any semantics based on this free type will be invalid.

Δ

Example 5

For a more rigorous argument of why a certain free type does not exist, consider a slightly simpler version of example 4, namely

$$Value ::= fun \langle\langle Value \rightarrow \{T, F\} \rangle\rangle$$

Here, the "base element" of *Value* is *fun* {} since {} is an element of $Value \rightarrow \{T,F\}$. But again, this free type does not exist because no function

$$fun : (Value \rightarrow \{T, F\}) \rightarrowtail Value$$

exists. The reason is as follows. The size of $Value \rightarrow \{T,F\}$ is $2^{\#Value}$, since each element of *Value* can be mapped to one of two values. By Cantor's theorem $\#Value < 2^{\#Value}$, for any set *Value* (even infinite). Thus no total injective function *fun* from $Value \rightarrow \{T,F\}$ to *Value* can possibly exist, and hence the free type *Value* does not exist.

Δ

Example 6

In Spivey [5], a rigorous argument is given which can be used to explain why the free type

$$T ::= c \langle\langle \mathbb{P}T \rangle\rangle$$

does not exist. The argument is as follows. Define the subset U of T, where

$$U = \{V : \mathbb{P}T \mid c\,V \notin V \bullet c\,V\}$$

Now for any set $S : \mathbb{P}T$

	$c\,S \in U$	
$\Leftrightarrow$	$\exists V : \mathbb{P}T \mid c\,V \notin V \bullet c\,V = c\,S$	[definition of U]
$\Leftrightarrow$	$\exists V : \mathbb{P}T \mid c\,V \notin V \bullet V = S$	[c is an injection]
$\Leftrightarrow$	$\exists V : \mathbb{P}T \bullet (c\,V \notin V) \wedge (V = S)$	[first order predicate calculus]
$\Leftrightarrow$	$\exists V : \mathbb{P}T \bullet (c\,S \notin S) \wedge (V = S)$	["]
$\Leftrightarrow$	$(c\,S \notin S) \wedge \exists V : \mathbb{P}T \bullet V = S$	["]
$\Leftrightarrow$	$c\,S \notin S$	[an existential witness for V is S]

So the following theorem has been derived

$$\vdash \forall S : \mathbb{P}T \bullet (c\,S \in U) \Leftrightarrow (c\,S \notin S) \qquad (3)$$

Specializing theorem (3) with $S = U$, the following contradictary theorem is obtained

$$\vdash (c\,U \in U) \Leftrightarrow (c\,U \notin U)$$

and so the free type T does not exist (an alternative argument for why T does not exist would be similar to that in example 5, since the size of $\mathbb{P}T$ is $2^{\#T}$).

Δ

Example 7

Now consider the free type

$$T ::= c \langle\langle \mathbb{F}T \rangle\rangle \qquad (4)$$

where $\mathbb{F}T$ is the set of all finite subsets of T. This time T exists, so it interesting to see where the contradiction in example 6 breaks down. If the reasoning of example 6 is followed, but with $\mathbb{F}$ replacing every occurrence of $\mathbb{P}$, then a theorem similar to (3) is obtained, namely

$$\vdash \forall S : \mathbb{F}T \bullet (c\,S \in U) \Leftrightarrow (c\,S \notin S) \qquad (5)$$

But this time, there is no guarantee that the set

$$U = \{V : \mathbb{F}T \mid c\,V \notin V \bullet c\,V\}$$

has type $\mathbb{F}T$ and so specializing theorem (5) with $S = U$ is not valid. The reason why U could be infinite, that is, not of type $\mathbb{F}T$, is as follows. Clearly, any set T which satisfies (4) is infinite. As T is infinite, U could be infinite, since there will be an infinite number of sets $V : \mathbb{F}T$ (remember if T is infinite then the set $\mathbb{F}T$ is infinite; it is just the elements of $\mathbb{F}T$, themselves sets, that are finite). Thus U could consist of an infinite number of $c\,V$.

Δ

1.2 Recursive functions

Having specified a recursive free type, the Z user will more than likely want to specify a recursive function over the free type. This use of recursive functions is another way inconsistencies can arise in Z specifications. Even if the recursive free type exists, the recursive function may not.

Example 8

The natural numbers can be considered as a free type, namely

$$nat ::= 0 \mid suc \langle\langle nat \rangle\rangle$$

Thus $nat = \{0, suc\ 0, suc(suc\ 0), ...\}$. The factorial function *!* below exists (where the abbreviation *1* has been used for *suc 0*)

$$\begin{array}{|l} _! : nat \rightarrow nat \\ \hline 0! = 1 \\ \forall\, n : nat \bullet (suc\ n)! = (suc\ n) \times n! \end{array}$$

But the function *f* below does not exist

$$\begin{array}{|l} f : nat \rightarrow nat \\ \hline f\,0 = 0 \\ \forall\, n : nat \bullet suc(f(suc\ n)) = (f\ n) \end{array}$$

since the second axiom gives $suc(f\ 1) = f\ 0$ (when $n = 0$). This together with the first axiom gives $suc(f\ 1) = 0$. Now $f\ 1$ can not possibly be an element of *nat*, for if it was then the equation $suc(f\ 1) = 0$ would contradict one of the axioms of the free type *nat*, namely

$$disjoint\ \langle \{0\}, ran\ suc \rangle$$

Even if the result of a function on the argument *1* is specified directly, for example

$$\begin{array}{|l} g : nat \rightarrow nat \\ \hline g\,0 = 0 \\ g\,1 = 0 \\ \forall\, n : nat \bullet g(suc\ n) = suc(g\ n) \end{array}$$

then this could lead to problems as well, since *g* does not exist either. The reason is that the third axiom gives $g\ 1 = suc(g\ 0)$ (when $n = 0$). This together with the first axiom gives $g\ 1 = 1$, which together with the second axiom gives $0 = 1$.

Δ

1.3 Content of the report

Two ways of proving that a Z recursive free type exists, are discussed. The first method is to prove the *finitary* condition for the free type. This is discussed in section 2.1, which also contains a strategy for proving the finitary condition. From this strategy it can be seen that a recursive free type T will exist provided that each arm $d \langle\!\langle E[T] \rangle\!\rangle$ that contains T is such that each element of $E[T]$ is formed from a finite number of elements of T. In particular, recursive free types containing only the constructions $\times$, $\mathbb{F}$, $\twoheadrightarrow$ and *seq* will exist.

The second method for proving that a recursive free type exists, discussed in section 2.2, is to use a *definitional extension*. The idea here is to construct a representation of the free type; the representation being a non-empty subset of an already existing type. The free type is then made isomorphic to its representation. The free type must then exist since it is isomorphic to a non-empty subset of an already existing type. The particular representation discussed is to use a set of labelled trees to represent the free type.

A technique for proving that a recursive function defined over a recursive free type exists, is also discussed. A theorem called the *primitive recursion theorem* (PRT) for the free type, is used to derive another theorem stating the existence of the function. Using the definitional extension method, the PRT can be proved from the representation; otherwise, having proved the finitary condition, the PRT may be stated as an axiom. The PRT is discussed in section 3. Section 4 contains a section on rules of thumb for the Z practitioner, on how to avoid writing inconsistent free types and recursive functions, as well as a summary and the conclusions of the report.

2 Proving recursive free types exist

The Z practitioner who is interested in some handy rules of thumb for avoiding inconsistent free types, rather than the details presented in this section, should go to section 4.1.

2.1 The finitary condition

In Spivey [3], a proof obligation is given which, if satisfied, means that the recursive free type exists. This condition is called the *finitary* condition, and is a sufficient, but not a necessary condition. The general form of a free type definition is

$$T ::= c1 \mid \ldots \mid cm \mid d1 \langle\!\langle E1[T] \rangle\!\rangle \mid \ldots \mid dn \langle\!\langle En[T] \rangle\!\rangle$$

where $E1[T]$, ... , $En[T]$ are expressions which might involve T. If any of them do involve T, then of course T is a recursive free type. T then exists provided that each $Ei[T]$ that does involve T, is a finitary construction of T. Roughly speaking, a construction is finitary if each element of it is built from a finite number of elements of T. In such cases, as an element of T is built from a finite number of other elements of T, each element of T can be "listed" in order (with respect to some ordering). The fact that the elements of T can be "listed" means that T must exist.

Example 9

The free type

$$T ::= a \mid b \langle\!\langle L \rangle\!\rangle \mid c \langle\!\langle M \times T \rangle\!\rangle \mid d \langle\!\langle N \mathrel{+\!\!\!\!\twoheadrightarrow} T \rangle\!\rangle$$

where L, M and N are given sets, will exist provided that the two constructions $M \times T$ and $N \mathrel{+\!\!\!\!\twoheadrightarrow} T$ are finitary. Now, each element of the construction $M \times T$ has the form (m, t) for some m in M and t in T, and so is built from one element of T. Thus $M \times T$ is finitary. Similarly $N \mathrel{+\!\!\!\!\twoheadrightarrow} T$ is finitary because each element of $N \mathrel{+\!\!\!\!\twoheadrightarrow} T$ has the form

$$\{n1 \mapsto t1, n2 \mapsto t2, \ldots, nk \mapsto tk\}$$

for some number k and $n1$, $n2$, ... , nk in N and $t1$, $t2$, ... , tk in T, and so consists of a finite number of elements of T; in this case k elements.

Δ

Formally, from Spivey [3], a construction $E[T]$ is a finitary construction of T, if for any countably infinite sequence of subsets

$$X_1 \subseteq X_2 \subseteq X_3 \subseteq \dots$$

of a set X, the following condition is satisfied

$$\bigcup(E[X_i]) = E[\bigcup(X_i)] \qquad (6)$$

This condition must be proved for any set X. The generalized unions $\bigcup$, are summing terms from 1 to infinity. Thus the left hand side (LHS) of (6), means

$$E[X_1] \cup E[X_2] \cup E[X_3] \cup \dots$$

and the right hand side (RHS) of (6) means

$$E[X_1 \cup X_2 \cup X_3 \cup \dots]$$

For any set S, the expression $E[S]$ is obtained from $E[T]$ by replacing all free occurrences of T in $E[T]$ by S. In Spivey [3], the finitary condition is stated slightly differently to (6). It is equivalent but also requires a construction $E[T]$ to be *monotonic*, that is, if $A \subseteq B$ then $E[A] \subseteq E[B]$. Condition (6) is the same as that stated in Arthan [6]. Arthan points out that any construction $E[T]$ satisfying (6) is also monotonic, and the proof of this is as follows.

Proof

Suppose $E[T]$ satisfies (6) and $A \subseteq B$. From this it must be shown that $E[A] \subseteq E[B]$. As (6) is true for any countably infinite sequence of subsets, then in particular it must be true for

$$A \subseteq B \subseteq B \subseteq B \subseteq \dots \qquad \text{(one } A\text{, the rest } B\text{)}$$

In this case (6) gives

$$E[A] \cup E[B] \cup E[B] \cup E[B] \cup \dots = E[A \cup B \cup B \cup B \cup \dots]$$

which can be simplified to

$$E[A] \cup E[B] = E[A \cup B] \qquad (7)$$

But $A \subseteq B$ and so $A \cup B = B$, hence from (7)

$$E[A] \cup E[B] = E[B]$$

From this, it must be the case that $E[A] \subseteq E[B]$ as required.

Δ

It is now instructive to see the finitary condition (6) proved for a particular construction.

Example 10

Consider the construction $E[T] = T \times T$. From (6), the following condition must be proved

$$\bigcup(X_i \times X_i) = \bigcup(X_i) \times \bigcup(X_i) \qquad (8)$$

for any countably infinite sequence of subsets $X_1 \subseteq X_2 \subseteq X_3 \subseteq \ldots$ of any set X. Notice that (8) is an equality between two sets. It can therefore be proved from the two statements

$$\bigcup(X_i \times X_i) \subseteq \bigcup(X_i) \times \bigcup(X_i) \qquad (9)$$
$$\bigcup(X_i \times X_i) \supseteq \bigcup(X_i) \times \bigcup(X_i) \qquad (10)$$

Statement (9) is the most straightforward and will be proved first. Let a be an element of the LHS. It must be shown that a is an element of the RHS. From the definition of $\bigcup$, if a is an element of the LHS then for some number n

$$a \in X_n \times X_n$$

Therefore

$$(fst(a) \in X_n) \wedge (snd(a) \in X_n)$$

Using the definition of $\bigcup$

$$(fst(a) \in \bigcup(X_i)) \wedge (snd(a) \in \bigcup(X_i))$$

and so

$$a \in \bigcup(X_i) \times \bigcup(X_i)$$

as required.

The proof of (10) is as follows. This time if a is an element of the RHS then it must be shown that a is an element of the LHS. If a is an element of the RHS then

$$(fst(a) \in \bigcup(X_i)) \wedge (snd(a) \in \bigcup(X_i))$$

From the definition of $\bigcup$, then for some m and n

$$(fst(a) \in X_m) \wedge (snd(a) \in X_n) \qquad (11)$$

Let max be the larger of the two numbers m and n. Then certainly

$$(m \leq max) \wedge (n \leq max) \qquad (12)$$

Now $X_1 \subseteq X_2 \subseteq X_3 \subseteq \ldots$, and so from (12)

$$(X_m \subseteq X_{max}) \wedge (X_n \subseteq X_{max}) \qquad (13)$$

From (11) and (13)

$$(fst(a) \in X_{max}) \wedge (snd(a) \in X_{max})$$

Thus

$$a \in X_{max} \times X_{max}$$

and so from the definition of $\bigcup$

$$a \in \bigcup(X_i \times X_i)$$

as required.

Δ

The proof of the finitary condition for other constructions is similar to above. Recall that the finitary condition is $\bigcup(E[X_i]) = E[\bigcup(X_i)]$, and this can be proved by proving the two conditions

$$\bigcup(E[X_i]) \subseteq E[\bigcup(X_i)] \qquad \text{(i)}$$
$$\bigcup(E[X_i]) \supseteq E[\bigcup(X_i)] \qquad \text{(ii)}$$

The proof of (i) is fairly straightforward, and the proof of (ii) involves constructing a number *max* as in the above proof of the finitary condition for $E[T] = T \times T$. The intuition behind *max* is as follows. The proof of (ii) is achieved by showing that if a is in the RHS then it is also in the LHS. Now if a is in the RHS then it is formed from elements of $\bigcup(X_i)$ (which shall, for the rest of this paragraph, be called the components of a). So each of these components must be in X_n for some number n. If $E[T]$ is finitary then there will only be a finite number of such n. The number *max* is the largest of these numbers n. Having obtained *max*, the proof of (ii) can then be completed, since all the components of a will be in X_{max} (as the X_i form an infinite sequence of subsets). Thus a can be constructed from elements of X_{max}, that is a is in $E[X_{max}]$ and so is in the LHS of (ii). A few examples of constructing *max* for various constructions $E[T]$ will now be given.

Example 11

Consider $E[T] = \mathbb{F}T$. (ii) above becomes

$$\bigcup(\mathbb{F}(X_i)) \supseteq \mathbb{F}(\bigcup(X_i))$$

To prove this, it has to be shown that if a is an element of the RHS, then a is also an element of the LHS. If a is an element of the RHS, then from the definition of $\mathbb{F}$, a is a subset of $\bigcup(X_i)$. So every element x of a is also an element of $\bigcup(X_i)$, and by the definition of $\bigcup$, is also an element of $X_{n(x)}$, for some number $n(x)$. This number is written as $n(x)$ to show its dependence on x. The number *max* is then the largest of the numbers $n(x)$, that is, the largest of the set of numbers

$$\{x : a \bullet n(x)\}$$

Δ

Example 12

Consider $E[T] = B \twoheadrightarrow T$, where B is a given set. This time (ii) above is

$$\bigcup(B \twoheadrightarrow X_i) \supseteq B \twoheadrightarrow \bigcup(X_i)$$

Once again it has to be shown that if a is an element of the RHS, then it is also an element of the LHS. If a is an element of the RHS, then $ran(a)$ is a subset of $\bigcup(X_i)$. So every element x of $ran(a)$ is also an element of $\bigcup(X_i)$, and by the definition of $\bigcup$, is also an element of $X_{n(x)}$, for some number $n(x)$. The number *max* is then the largest of the numbers $n(x)$, that is, the largest of the set of numbers

$$\{x : ran(a) \bullet n(x)\}$$

Δ

Example 13

As a final example of finding *max*, consider $E[T] = T \times \mathbb{F}T$ which contains both $\times$ and $\mathbb{F}$. This time (ii) above is

$$\bigcup(X_i \times \mathbb{F}(X_i)) \supseteq \bigcup(X_i) \times \mathbb{F}(\bigcup(X_i))$$

Once again it has to be shown that if a is an element of the RHS, then it is also an element of the LHS. If a is an element of the RHS, then $fst(a)$ is an element of $\bigcup(X_i)$ and $snd(a)$ is an element of $\mathbb{F}(\bigcup(X_i))$. So by the definition of $\bigcup$, $fst(a)$ is an element of X_m for some number m, and from the definition of $\mathbb{F}$, $snd(a)$ is a subset of $\bigcup(X_i)$. So every element x of $snd(a)$ is also an element of $\bigcup(X_i)$, and by the definition of $\bigcup$, is also an element of $X_{n(x)}$, for some number $n(x)$. The number *max* is then the largest of the set of numbers

$$\{m\} \cup \{x : snd(a) \bullet n(x)\}$$

Δ

So it can be seen that a construction $E[T]$ is finitary if each element of $E[T]$ is made from a finite number of elements of T. In particular, constructions of T just involving $\times$, $\mathbb{F}$, $\twoheadrightarrow$ and *seq* will be finitary, for example

$$(A \times T) \twoheadrightarrow (seq\ T)$$

where A is a given set. From examples 10, 11, 12 and 13 it can be seen that the formation of *max* in such cases could be automated. Thus some, if not all, of the proof of the finitary condition for a recursive free type involving only $\times$, $\mathbb{F}$, $\twoheadrightarrow$ and *seq* could be automated.

2.1.1 The finitary condition and the world of sets

In Spivey[4], a semantics of the Z language is given. This semantics is in terms of a world of sets, *W*, in which everything is a set. The idea is that the meaning of each piece of Z can be explained by giving it a representation in *W*. The relationship between a piece of Z and its representation is known as a *model* for the piece of Z. The axioms of *W* are those of Zermelo-Fraenkel set theory, but with the axioms of *replacement* and *choice* omitted. But there has been some discussion recently as to whether all finitary free types have a model in *W*, and hence whether a semantics can be given for them. It is not obvious that every finitary free type has a model in *W*. But Arthan [6] has shown that if the axiom of choice is added to the axioms of *W* then it is certain that every finitary free type has a model in *W*.

2.2 Definitional extension

Another way to be sure that a recursive free type exists is to use a *definitional extension*. A definitional extension is where a new object is defined in terms of existing objects, in a way that ensures the existence of the new object. In the case of free types, this means defining a free type in terms of a subset of an existing type. The subset is identified by supplying a predicate over the existing type. The subset consists of all those elements of the existing type which satisfy the predicate. There is a proof obligation that the subset is non-empty. The free type is then simply defined to be isomorphic to the subset. The constructors of the free type are then defined.

The subset described above is thus a representation of the free type, and the trick is to find the right representation that truly captures the semantics of the free type. It will soon become obvious this is not the case, if the usual properties of the free type can not be proved from the representation (for example that the constructors are injective). The work presented here is based on Melham [2], but has been extended to deal with more complicated free types. Melham's work can only be used to show how a free type T involving existing types, $\times$ and simple occurrences of T, for example

$$T ::= c \langle\!\langle A \times T \rangle\!\rangle \mid d \langle\!\langle B \times T \times T \rangle\!\rangle$$

where A and B are given sets, can be represented. It can not be used for example, to show how the free type

$$T ::= c \langle\!\langle A \mathrel{+\!\!\!\!\twoheadrightarrow} T \rangle\!\rangle$$

can be represented. The reason why Melham did not consider more complicated free types is so that the work could be easily automated in the HOL theorem proving system [1]. Both the representation of the free type and the proof of the primitive recursion theorem (used to prove the existence of recursive functions over recursive free types, see section 3) has been automated in HOL. The ML function in HOL which does this is called *define_type*. This section first describes Melham's work and then shows how it can be extended.

In Melham's work, a set of labelled finite trees is used to represent a free type. The actual labels used and the shape of the trees depends on the particular free type. Labelled trees can themselves be defined using a definitional extension. The type used to represent labelled trees can also be defined using a definitional extension, and so on. In fact, any new type in HOL can be built up from existing types using a definitional extension. Melham's work for free types is best expained by an example. The following example is explained using Z, but the annex shows how *define_type* in HOL automatically constructs the representation. The annex also shows how *define_type* automatically proves the primitive recursion theorem for the free type.

Example 14

Consider the free type

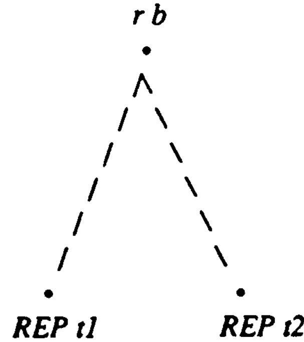
$$T ::= c \langle\!\langle A \rangle\!\rangle \mid d \langle\!\langle B \times T \times T \rangle\!\rangle$$

where A and B are given sets. An element of T can be represented by a tree labelled with elements of *Tlabels* where

$$Tlabels ::= l \langle\!\langle A \rangle\!\rangle \mid r \langle\!\langle B \rangle\!\rangle$$

Note that the free type *Tlabels* exists as it is simply the labelled disjoint union of A and B, and is not recursive. In general, *Tlabels* will have the same number of arms as T. Let *Tlabels_ltree* denote the type of trees of *any* shape whose nodes are labelled with elements of *Tlabels*. The free type T is to be represented by a subset of *Tlabels_ltree*. An element $c\ a$ of T, can be represented by the tree

An element $d(b,t1,t2)$ of T, where b is an element B and $t1$, $t2$ are elements of T can be represented by the tree



where *REP t1* and *REP t2* are the tree representations of $t1$ and $t2$. Let *Node label subtree_seq* denote the tree with top node labelled with *label*, and sequence of subtrees *subtree_seq* from that node. Then basically, a tree will represent an element of T provided that

$$(\exists\, a : A \bullet label = l\ a) \wedge (\#\, subtree_seq = 0)$$
$$\vee\ (\exists\, b : B \bullet label = r\ b) \wedge (\#\, subtree_seq = 2)$$

Actually, as it stands, the above predicate only describes the top of such a tree. For example, there will be trees satisfying the above predicate which contain a node (lower down the tree) with three or more subtrees branching from it. But such trees do not represent elements of T. For this reason, the above predicate is strengthened by applying a function *TRP* (see annex). This function makes sure that the above predicate holds all the way down the tree. The resulting predicate then characterises the subset of *Tlabels_ltree* which is to represent T. Next, the free type T is simply defined to be isomorphic to the subset, giving the isomorphisms *REP* of type $T \rightarrow Tlabels_ltree$ and *ABS* of type $Tlabels_ltree \rightarrow T$. The constructors, c and d of T can now be defined:

$$\forall\, a : A \bullet c\ a = ABS(Node\ (l\ a)\ \langle\rangle)$$
$$\forall\, b : B;\ t1, t2 : T \bullet d(b,t1,t2) = ABS(Node\ (r\ b)\ \langle REP\ t1, REP\ t2\rangle)$$

Δ

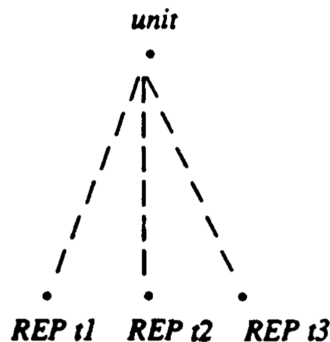
The next three examples show how Melham's technique can be extended. To keep some uniformity, labelled trees will be used throughout to represent the following three free types. All three examples are explained using Z.

Example 15

Consider the free type

$$T ::= c \langle\langle seq\ T \rangle\rangle$$

This time let *Tlabels* be a type consisting of a single value, say *unit*, and *Tlabels_ltree* be the type of trees, of any shape, whose nodes are labelled with *unit*. An element $c\ s$ of T where s is an element of $seq\ T$ can be represented by the tree whose top node is labelled with *unit* and with a subtree for each element of s; the subtree being the representation of the element of s. For example, the element $c\ \langle t1, t2, t3\rangle$ of T is represented by



In fact, every element of *Tlabels_ltree* will represent some element of T. Thus T is represented by the whole of *Tlabels_ltree*. So the predicate which characterises the required subset of *Tlabels_ltree* is simply *true*. The free type T is then defined to be isomorphic to the whole of *Tlabels_ltree* and the constructor c is then defined as

$$\forall\ s : seq\ T \bullet c\ s = ABS(Node\ unit\ (map\ REP\ s))$$

where *map REP s* is the sequence consisting of the representations of the elements of s.

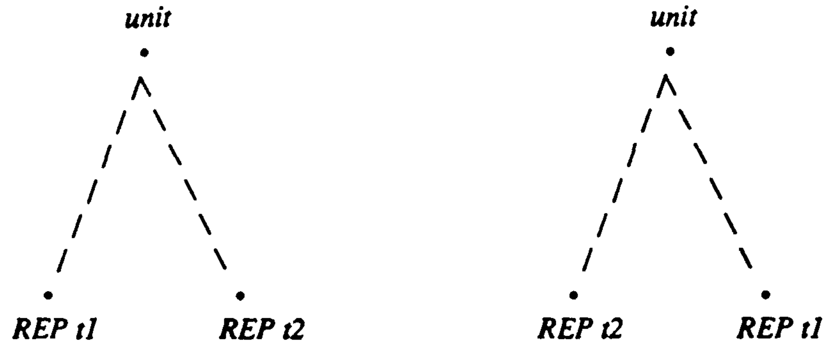
Δ

Example 16

Consider the free type

$$T ::= c \langle\!\langle \mathbb{F} T \rangle\!\rangle$$

Let *Tlabels* and *Tlabels_ltree* be those of the last example. This time, only a subset of *Tlabels_ltree* will represent T. This is because distinct elements such as $c \langle t1, t2 \rangle$ and $c \langle t2, t1 \rangle$ in the last example, collapse down to the single element $c \{t1, t2\}$ in this example. Hence, loosely speaking, the free type in this example does not have so many elements as the free type in the last example, and so its representation will not have so many elements. Consider an element $c \{t1,t2\}$ of T. As $c \{t1,t2\} = c \{t2,t1\}$ then which one of the two trees



should be used as the representation? The problem is overcome by defining a function

$$set_seq : \mathbb{F} X \rightarrow seq\ X$$

which converts a set into a sequence; it orders the elements of the set. The particular ordering produced by *set_seq* is not important, only the fact that that they are ordered. The function *set_seq* is a polymorphic function and so can be applied to a set of anything. This function can be used to determine which of the two trees above should be used as the representation of $c\,\{t1, t2\}$. Suppose $set_seq\,\{REP\ t1, REP\ t2\} = \langle REP\ t2, REP\ t1\rangle$, then the right hand tree above will be used as the r-presentation of $c\,\{t1, t2\}$. Basically, a tree *Node label subtree_seq* will represent an ele..ent of T provided that the following predicate holds

$$subtree_seq \in (ran\ set_seq)$$

Once again, as explained in example 14, the function *TRP* must be applied to the above predicate, to give the actual predicate which characterises the required subset of *Tlabels_ltree*. Once again, T is then defined to be isomorphic to this subset. The constructor c is then defined as

$$\forall\, s : \mathbb{F}T \bullet c\ s = ABS(Node\ unit\ (set_seq(REP\ \llparenthesis s \rrparenthesis)))$$

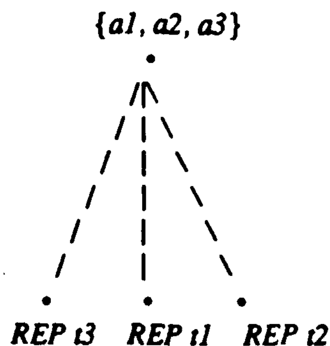
Δ

Example 17

Consider the free type

$$T ::= c \langle\langle A \twoheadrightarrow\!\!\!\!\!\!+ T \rangle\rangle$$

where A is a given set. This time let $Tlabels = \mathbb{F}A$, and as usual *Tlabels_ltree* be the type of trees, of any shape, labelled with elements of *Tlabels*. An element $c\,f$ of T can be represented by the tree whose top node is labelled $dom\ f$ and which has a subtree representing each $f\ a$, where a is in $dom\ f$. The order of these subtrees can again be determined by the function *set_seq* which appeared in the last example. For example, consider the element $c\,\{a1 \mapsto t1, a2 \mapsto t2, a3 \mapsto t3\}$ of T. Suppose $set_seq\,\{a1, a2, a3\} = \langle a3, a1, a2\rangle$. Then this element of T can be represented by



An element of *Tlabels_ltree* will represent an element of *T* provided that, at every node, the cardinality of the label is equal to the number of subtrees from that node. So, basically an element *Node label subtree_seq* of *Tlabels_ltree* is a representation if

$$\#label = \#subtree_seq$$

Once again, as explained in example 14, the function *TRP* must be applied to the above predicate. After defining *T* to be isomorphic to the required subset of *Tlabels_ltree*, the constructor *c* can be defined as

$$\forall f : A \twoheadrightarrow\!\!\!\!\!\!+\; T \bullet c\,f = ABS(Node\ (dom\,f)\ (map\ (REP \circ f)\ (set_seq(dom\,f))))$$

Δ

2.2.1 Problems with using definitional extensions

Recall that using definitional extensions to define free types involves a proof obligation that the subset of the existing type is non-empty. The free type is then made isomorphic to this subset. This proof obligation only ensures that the free type is non-empty. It does not ensure that the free type has the intended semantics. For example, consider the free type $T ::= c \langle\!\langle \mathbb{F} T \rangle\!\rangle$ in example 16. Suppose the mistake was made, that the predicate characterising the subset of *Tlabels_ltree* was too strong, so that only the tree

unit

•

satisfied it. Then this subset of one tree certainly satisfies the proof obligation, but the free type would then only have one element, since the free type is made isomorphic to this subset. The free type would then not have the semantics of *T*, since *T* has an infinite number of elements.

Another problem with using definitional extensions is when the free type is complicated. For example, what representation should be used for the free type $T ::= c \langle\!\langle \mathbb{F}\, \mathbb{F}\, T \rangle\!\rangle$? If labelled trees are used as the representation of such free types, then the representation certainly will not be as neat as those discussed so far.

3 Proving recursive functions exist

This section discusses a way of proving that a recursive function, defined over a recursive free type, exists. The Z practitioner who is interested in some handy rules of thumb for avoiding inconsistent recursive functions, rather than the details presented here, should go to section 4.2. The technique described in this section is to use the *primitive recursion theorem* (PRT) for the free type, to prove another theorem which states the existence of the function. The PRT captures the semantics of the free type, but in a way that allows the existence of recursive functions to be proved. The PRT can be used to prove the existence of a function f specified by *primitive recursion* on a free type T. That is, for any arm $d \langle\langle E[T] \rangle\rangle$ of T which contains T, then $f(d\ x)$ where x is an element of $E[T]$, is specified in terms of an expression involving f and x. The following examples will make this clear. The PRTs in the following examples can all be proved from a representation of the particular free type; this representation coming from the definitional extension method described in section 2.2. The annex gives an example of this, showing how the construction of the representation and the proof of the PRT is carried out in the HOL system. Also, as the PRT captures the semantics of the free type, then not surprisingly all the usual properties of a free type can be proved from it. For example, the PRT can be used to prove that the elements of a free type T that can be generated from the arms of T are the only elements of T; that is they exhaust T. The next example, which proves the existence of the factorial function, $!$, over the natural numbers, also gives some intuition into the PRT.

Example 18

The natural numbers can be considered as a free type, namely

$$nat ::= 0 \mid suc \langle\langle nat \rangle\rangle$$

The PRT for nat is

$$\begin{aligned} \vdash\ & \forall\, e : X; f : (X \times nat) \rightarrow X \bullet \\ & \quad \exists_1\, h : nat \rightarrow X \bullet \\ & \quad\quad h\ 0 = e \\ & \quad\quad \wedge\ \forall\, n : nat \bullet h(suc\ n) = f(h\ n, n) \end{aligned}$$

The theorem is generic in X. The PRT looks a bit strange at first, but it is simply saying that each e and f define a recursive function h (for example h could be the factorial function); e is the base case and f is the body of h. The PRT captures the fact, for example, that the elements of nat generated by its two arms, exhaust nat. The reason is as follows. Notice that the function h in the PRT is unique once defined on each arm of nat. If nat contained any more elements than those generated by its two arms, then these extra elements could be mapped by h in a number of different ways, yielding a different function in each case. This would contradict h being unique.

The PRT can be used, for example, to prove that the factorial function

$$
\begin{array}{|l}
_! : nat \rightarrow nat \\
\hline
0! = 1 \\
\forall\, n : nat \bullet (suc\ n)! = (suc\ n) \times n!
\end{array}
\qquad (14)
$$

which is recursive, actually exists. The abbreviation *1* has been used for the element *suc 0* of *nat*. The existence theorem for *!* is proved as follows. The idea is to instantiate the generic set X, and specialize e and f in the PRT, so that the function h in the PRT becomes the factorial function. So instantiating X to be *nat*, and then specializing e to be *1* and f to be

$$\lambda\, x, y : nat \bullet (suc\ y) \times x$$

the PRT gives

$$
\begin{aligned}
\vdash \exists_1\, h : nat \rightarrow nat \bullet \\
&h\, 0 = 1 \\
\wedge\ &\forall\, n : nat \bullet h(suc\ n) = (\lambda\, x, y : nat \bullet (suc\ y) \times x)\,(h\, n, n)
\end{aligned}
$$

The RHS of the second equality can be simplified by β-reduction (function application) to give

$$
\begin{aligned}
\vdash \exists_1\, h : nat \rightarrow nat \bullet \\
&h\, 0 = 1 \\
\wedge\ &\forall\, n : nat \bullet h(suc\ n) = (suc\ n) \times (h\, n)
\end{aligned}
$$

This theorem says that the factorial function specified in (14) above, exists. It is interesting to note that this theorem also says that the factorial function is unique, and so could have been specified as

$$
\begin{array}{|l|}
\hline\hline
_! : nat \rightarrow nat \\
\hline
0! = 1 \\
\forall\, n : nat \bullet (suc\ n)! = (suc\ n) \times n! \\
\hline
\end{array}
$$

Δ

Example 19

Consider the free type

$$T ::= c \langle\!\langle A \rangle\!\rangle \mid d \langle\!\langle B \times T \rangle\!\rangle$$

where A and B are given sets. The PRT for T is

$$\begin{array}{l} \vdash \forall f : A \rightarrow X;\ g : (X \times B \times T) \rightarrow X \bullet \\ \quad \exists_1 h : T \rightarrow X \bullet \\ \qquad \forall a : A \bullet h(c\ a) = f\ a \\ \qquad \wedge\ \forall b : B;\ t : T \bullet h(d(b,t)) = g\ (h\ t, b, t) \end{array} \tag{15}$$

The PRT can be used to prove that the following function *base* exists (which computes the base element of a member of T).

$$\begin{array}{|l} base : T \rightarrow T \\ \hline \forall a : A \bullet base(c\ a) = c\ a \\ \forall b : B;\ t : T \bullet base(d(b,t)) = base\ t \end{array}$$

If the PRT (15) is first instantiated with the generic set X taking the value T, and then specialized with the function f taking the value

$$\lambda\ a : A \bullet c\ a$$

and the function g taking the value

$$\lambda\ t1 : T;\ b : B;\ t2 : T \bullet t1$$

the theorem

$$\begin{array}{l} \vdash \exists_1 h : T \rightarrow T \bullet \\ \qquad \forall a : A \bullet h(c\ a) = (\lambda\ a : A \bullet c\ a)\ a \\ \qquad \wedge\ \forall b : B;\ t : T \bullet h(d(b,t)) = (\lambda\ t1 : T;\ b : B;\ t2 : T \bullet t1)\ (h\ t, b, t) \end{array}$$

is obtained. This theorem can then be simplified by β-reduction to give

$$\begin{array}{l} \vdash \exists_1 h : T \rightarrow T \bullet \\ \qquad \forall a : A \bullet h(c\ a) = c\ a \\ \qquad \wedge\ \forall b : B;\ t : T \bullet h(d(b,t)) = h\ t \end{array}$$

Hence the function *base* certainly exists. Once again it is unique, and so could be specified as

$$base : T \rightarrow T$$

$$\forall a : A \bullet base(c\ a) = c\ a$$
$$\forall b : B; t : T \bullet base(d(b,t)) = base\ t$$

Δ

Example 20

The PRT can also be used to prove the existence of a recursive function that has only been specified on some of the arms of the free type (that is, underspecified). For example, the function

$$\gamma : nat \rightarrow nat$$

$$\forall\ n : nat \bullet \gamma(suc\ n) = 2 \times (\gamma\ n)$$

defined over the free type of natural numbers in example 18, has only been specified on the second arm of *nat*. The abbreviation *2* has been used for the element *suc(suc 0)* of *nat*. There are many functions which satisfy the above specification, each giving a different value for $\gamma\ 0$. So any existence theorem for γ will state simple existence, rather than unique existence. To obtain the existence theorem, the PRT for *nat*, which appears in example 18 is first instantiated with *X* taking the value *nat*, and then specialized with *e* taking the value *0* (in fact, this could be any value of *nat*), and *f* taking the value

$$\lambda\ x, y : nat \bullet 2 \times x$$

followed by β-reduction to obtain

$$\vdash \exists_1\ h : nat \rightarrow nat \bullet$$
$$h\ 0 = 0$$
$$\wedge\ \forall\ n : nat \bullet h(suc\ n) = 2 \times (h\ n)$$

This theorem can be weakened to give

$$\vdash \exists\, h : nat \rightarrow nat \bullet \\ \quad h\,0 = 0 \\ \quad \wedge\ \forall\, n : nat \bullet h(suc\ n) = 2 \times (h\ n)$$

(the $\exists_1$ has been replaced by $\exists$), which in turn can be used to derive

$$\vdash \quad h\,0 = 0 \\ \quad \wedge\ \forall\, n : nat \bullet h(suc\ n) = 2 \times (h\ n)$$

for some $h : nat \rightarrow nat$. From this latest theorem, it follows that

$$\vdash \forall\, n : nat \bullet h(suc\ n) = 2 \times (h\ n)$$

and so

$$\vdash \exists\, h : nat \rightarrow nat \bullet \\ \quad \forall\, n : nat \bullet h(suc\ n) = 2 \times (h\ n)$$

which is the required existence theorem.

Δ

Example 21

Consider the function

$$\begin{array}{|l} \delta : nat \rightarrow nat \\ \hline \forall\, n : nat \bullet \delta(suc\ n) \in \{m : nat \bullet m \times (\delta\ n)\} \end{array}$$

defined over the free type of natural numbers in example 18. The function δ is even more underspecified than γ in example 20. This time, not only has δ just been specified on the second arm of *nat*, but it is underspecified on this arm. The specification of δ can be strengthened to

$$\begin{array}{|l} \delta_1 : nat \rightarrow nat \\ \hline \forall\, n : nat \bullet \delta_1(suc\ n) = 2 \times (\delta_1\ n) \end{array}$$

Any δ_1 which satisfies this new specification will also satisfy the specification of δ. The specification of δ_1 is the same as the specification of γ in example 20, where it was shown that γ existed. Hence δ_1 exists and thus so does δ.

Δ

Example 22

Another example of a PRT, is that for the free type $T ::= c \langle\langle\, seq\ T \,\rangle\rangle$, which is

$$\vdash \forall f : (seq\ X \times seq\ T) \rightarrow X \bullet$$
$$\exists_1\ h : T \rightarrow X \bullet$$
$$\forall\ s : seq\ T \bullet h(c\ s) = f(map\ h\ s, s)$$

where *map h s* is the new sequence formed from *s* by applying the function *h* to each element of *s*. For example, if *square* is the function which squares a number, then *map square* $\langle 2, 1, 5 \rangle = \langle 4, 1, 25 \rangle$. The reason why the expression *map h s* is required in the PRT above, is as follows. The function *h* is defined by primitive recursion. Thus, *h(c s)* will be defined in terms of an expression involving *h* applied to *every* element of *T* that directly makes up the element *c s* of *T*. These elements of *T* that make *c s* are the elements of *s*. Hence *h* must be applied to every element of *s*; hence the expression *map h s*.

Δ

Example 23

Another example is the PRT for the free type $T ::= c \langle\langle\, \mathbb{F} T \,\rangle\rangle$ which is

$$\vdash \forall f : (\mathbb{F} X \times \mathbb{F} T) \rightarrow X \bullet$$
$$\exists_1\ h : T \rightarrow X \bullet$$
$$\forall\ set : \mathbb{F} T \bullet h(c\ set) = f(h(\!|set|\!), set)$$

Δ

As mentioned at the start of this section, the PRTs shown so far can all be proved from a representation of the particular free type; this representation coming from the definitional extension method as described in section 2.2. The proof of the PRT depends on the particular representation, but the author conjectures that the PRT for a general free type

$$T ::= c1 \mid \ldots \mid cm \mid d1 \langle\!\langle E1[T] \rangle\!\rangle \mid \ldots \mid dn \langle\!\langle En[T] \rangle\!\rangle \qquad (16)$$

is

$$\begin{array}{l} \vdash \forall\, e1, \ldots em : X; \qquad (17) \\ \quad f1 : (E1[X] \times E1[T]) \to X; \ldots fn : (En[X] \times En[T]) \to X \bullet \\ \quad\quad \exists_1\, h : T \to X \bullet \\ \quad\quad\quad h\, c1 = e1 \wedge \\ \quad\quad\quad \ldots \\ \quad\quad\quad h\, cm = em \wedge \\ \quad\quad\quad \forall\, x : E1[T] \bullet h(d1\, x) = f1\, (x', x) \wedge \\ \quad\quad\quad \ldots \\ \quad\quad\quad \forall\, x : En[T] \bullet h(dn\, x) = fn\, (x', x) \end{array}$$

where x' is obtained from x by replacing any $t \in T$ appearing in x, by $h\ t$.

To see how the PRT for a particular free type can be derived from (17), consider the next example.

Example 24

Consider the free type in example 19, namely

$$T ::= c \langle\!\langle A \rangle\!\rangle \mid d \langle\!\langle B \times T \rangle\!\rangle$$

Comparing T with the general form for a free type (16) yields

$$m = 0,\ n = 2,\ d1 = c,\ d2 = d,\ E1[T] = A,\ E2[T] = B \times T$$

Therefore $E1[X] = A$ and $E2[X] = B \times X$. The general form for the PRT (17) therefore gives

$$\begin{array}{l} \vdash \forall\, f1 : (A \times A) \to X; f2 : ((B \times X) \times (B \times T)) \to X \bullet \qquad (18) \\ \quad \exists_1\, h : T \to X \bullet \\ \quad\quad \forall\, x : A \bullet h(c\, x) = f1\, (x', x) \wedge \\ \quad\quad \forall\, x : (B \times T) \bullet h(d\, x) = f2\, (x', x) \end{array}$$

Next, the x' are eliminated as described above. For any $x : A$, there are no elements of T present in x, and so $x' = x$. For $x : (B \times T)$, $x = (b,t)$ for some $b : B$ and $t : T$, and so $x' = (b, h\ t)$. Thus (18) may be rewritten

$$\begin{array}{l} \vdash \forall\, f1 : (A \times A) \to X; f2 : ((B \times X) \times (B \times T)) \to X \bullet \qquad (19) \\ \quad \exists_1\, h : T \to X \bullet \\ \quad\quad \forall\, x : A \bullet h(c\, x) = f1\, (x, x) \wedge \\ \quad\quad \forall\, b : B; t : T \bullet h(d(b,t)) = f2\, ((b, h\, t), (b,t)) \end{array}$$

Theorem (19) can be made exactly the same as theorem (15) (the PRT for T in example 19) by specializing it with the function $f1$ taking the value

$$\lambda\, x, y : A \bullet f\, x$$

and $f2$ taking the value

$$\lambda\, x : (B \times X); y : (B \times T) \bullet g\ (snd\, x, fst\, y, snd\, y)$$

followed by β-reduction to give

$$\begin{aligned} \vdash \exists_1\, h : T \rightarrow X \bullet \\ \forall\, x : A \bullet h(c\, x) = f\, x \wedge \\ \forall\, b : B; t : T \bullet h(d(b,t)) = g\ (h\, t, b, t) \end{aligned} \qquad (20)$$

The functions f and g in theorem (20) can then be generalized, followed by renaming the bound variable x to be a, to yield theorem (15).

Δ

3.1 Proving a primitive recursion theorem

Recall that in section 2, two methods were given to prove that a recursive free type existed. The first was to prove the finitary condition; the second was to use a definitional extension and construct a representation for the free type. Proving the finitary condition just means that the free type exists; the condition itself contains no semantics of the free type (for example that the constructors are injective). The PRT for the free type can therefore not be proved directly from the finitary condition. Using a definitional extension, the PRT can be proved using the representation of the free type; an example of this can be found in the annex.

4. Summary and conclusions

Sections 4.1 and 4.2 contain some rules of thumb for Z practitioners, on how to avoid writing inconsistent free types and recursive functions. Section 4.3 contains the general summary and conclusions of the report.

4.1 Free types

The following method can be used to see if a recursive free type T exists. Firstly identify each arm of the free type that involves T. Then, for each such arm, check that every element of the expression inside the angled brackets $\langle\!\langle\rangle\!\rangle$, is made from a finite number of elements of T. Also, recall from the introduction that T must have one or more "base elements"; elements to allow other more complicated elements to be built up.

Example 25

Consider the free type

$$T ::= a \mid b \langle\!\langle A \rangle\!\rangle \mid c \langle\!\langle B \times T \rangle\!\rangle \mid d \langle\!\langle C \mathrel{+\!\!\!\!\!\twoheadrightarrow} T \rangle\!\rangle \mid e \langle\!\langle \mathit{seq}\, T \rangle\!\rangle$$

where A, B and C are given sets. There are three arms of this free type that contain T, namely

$$\begin{array}{ll} c \langle\!\langle B \times T \rangle\!\rangle & \text{(i)} \\ d \langle\!\langle C \mathrel{+\!\!\!\!\!\twoheadrightarrow} T \rangle\!\rangle & \text{(ii)} \\ e \langle\!\langle \mathit{seq}\, T \rangle\!\rangle & \text{(iii)} \end{array}$$

Each element of the expression $B \times T$ in (i) has the form (b, t) for some b in B and t in T, and is thus made from a finite number of elements of T; namely one. Each element of the expression $C \mathrel{+\!\!\!\!\!\twoheadrightarrow} T$ in (ii) has the form

$$\{c1 \mapsto t1, c2 \mapsto t2, \dots, ck \mapsto tk\}$$

for some number k and $c1, c2, \dots, ck$ in C and $t1, t2, \dots, tk$ in T. Each element is thus made from a finite number of elements of T; in this case k. Finally, each element of the expression $\mathit{seq}\, T$ in (iii) has the form

$$\langle t1, t2, \dots, tn \rangle$$

for some number n and $t1, t2, \dots, tn$ in T. Each element is thus made from a finite number of elements of T; in this case n. Also the "base elements" come from the first two arms of T. The free type T therefore exists.

Δ

Example 26

Consider the free type

$$T ::= c \mid d \langle\langle S \rangle\rangle$$

where S is the schema

$$\begin{array}{|l} \hline S \\ a : A \\ t : T \\ \hline \end{array}$$

with A a given set. So just the arm $d \langle\langle S \rangle\rangle$ of T contains T. If the following notation is used to denote an element of S

$$\{'a' \mapsto a, 't' \mapsto t\}$$

which states that the strings $'a'$ and $'t'$ are bound to particular values of A and T, then each element of S is made from exactly one element of T. Also, the first arm of T contains the "base element" c. The free type T therefore exists.

Δ

4.2 Recursive functions

Given a recursive free type T that exists, the primitive recursion theorem (PRT) for T can be used to see if a recursive function f, defined over T, exists. This is fully explained in section 3. Basically, the PRT is used to try and produce a theorem stating the existence of f. If the following two simple rules for specifying f are followed, then the attempt to produce an existence theorem is more likely to be successful. First, specify f on each arm of T separately. The function f does not have to be specified on every arm of T. Secondly, for any arm $d \langle\langle E[T] \rangle\rangle$ of T that contains T, specify $f(d\,x)$, where x is an element of $E[T]$, by primitive recursion. That is, specify $f(d\,x)$ in terms of an expression involving f and x. The following examples will make these two rules clear.

Example 27

The natural numbers can be considered as a free type, namely

$$nat ::= 0 \mid suc \langle\langle nat \rangle\rangle$$

Now consider the factorial function *!*. There are two arms in the free type definition of *nat*, namely *0* and *suc* ⟪ *nat* ⟫, and so a recursive function over *nat* could be specified on each arm separately, or just the second arm. The function *!* is specified on each arm as below. The abbreviation *1* has been used for *suc 0*. As the second arm of *nat* contains *nat*, then *(suc n)!* is specified in terms of *n!*.

$$_! : nat \rightarrow nat$$

$$0! = 1$$
$$\forall\, n : nat \bullet (suc\ n)! = (suc\ n) \times n!$$

Δ

Example 28

Consider the free type

$$T ::= c \langle\langle L \rangle\rangle \mid d \langle\langle M \times T \rangle\rangle \mid e \langle\langle T \times N \rangle\rangle$$

where *L*, *M* and *N* are given sets. Now consider the function *f* as specified below. The function is specified on just the first two arms, but separately. Also, as the second arm of *T* contains *T*, then *f(d(m,t))* is specified in terms of *f t*.

$$f : T \rightarrow L$$

$$\forall\, l : L \bullet f(c\ l) = l$$
$$\forall\, m : M; t : T \bullet f(d(m,t)) = f\ t$$

Δ

4.3 General summary and conclusions

One method of proving that a recursive free type exists, is to prove the finitary condition for that free type, as discussed in section 2.1. This section describes a strategy for proving the finitary condition. From this strategy it can be seen that a recursive free type will exist provided that each arm $d \langle\!\langle E[T] \rangle\!\rangle$ of T that contains T, is such that every element of $E[T]$ consists of a finite number of elements of T. So for example, a recursive free type made from just $\times$, $\mathbb{F}$, $\twoheadrightarrow\!\!\!\!\!\!+$ and *seq* will exist. So by simply inspecting a free type definition by eye, its existence can, in some cases, be asserted. Some examples of this are given in section 4.1. In other cases, the existence of the free type will not be so obvious, and the finitary condition will have to be proved. The finitary proof obligation can be automatically produced. It is not obvious how much of the proof can be automated, but following the strategy in section 2.1 will lead to a proof. From the strategy, it can be seen that if the free type contains only $\times$, $\mathbb{F}$, $\twoheadrightarrow\!\!\!\!\!\!+$ and *seq*, then most, if not all of the proof (if a proof was required), could be automated. Having established the existence of the free type, by proving the finitary condition, the primitive recursion theorem (PRT) for the free type can then be asserted as an axiom. The PRT can then be used to prove that a recursive function defined over the free type exists, as discussed in section 3.

Another method for proving that a recursive free type exists is to use a definitional extension, as discussed in section 2.2. The idea here is to construct a representation for the free type. The particular representation discussed, is to use a set of labelled trees to represent the free type. The free type is then made isomorphic to its representation. The PRT for this free type can then be proved using its representation. The PRT can then be used to prove that a recursive function defined over the free type exists, just as before. It is not obvious whether the construction of the representation can be automated. Also, the representation itself could get a bit complicated. For example, what representation should be used for $T ::= c \langle\!\langle \mathbb{F}\,\mathbb{F}\,T \rangle\!\rangle$? Also it is not obvious how much of the proof of the PRT using the representation can be automated. Certainly the construction of the representation, and proof of the PRT for a free type definition, T, consisting only of existing types, $\times$ and simple occurrences of T, for example

$$T ::= c \langle\!\langle A \times T \rangle\!\rangle \mid d \langle\!\langle B \times T \times T \rangle\!\rangle$$

where A and B are given sets, can be fully automated. The automation of such free types would be analagous to the automation in Melham's type definition package [2] in HOL.

The process of trying to obtain an existence theorem for a recursive function f from a PRT for a free type T, is more likely to succeed if the two simple rules described in section 4.2 are followed. The first rule is that f should be defined on each arm of T separately, but f does not have to be defined on every arm of T. The second rule is that for any arm $d \langle\!\langle E[T] \rangle\!\rangle$ of T that contains T, specify $f(d\,x)$, where x is an element of $E[T]$, by primitive recursion. That is, specify $f(d\,x)$ in terms of an expression involving f and x. The examples in section 4.2 clarify these two rules.

References

1. *The HOL System (Description)*, SRI International, Cambridge Research Centre, December 1989.

2. T. F. Melham, "Automating Recursive Type Definitions in Higher Order Logic", in *Current Trends in Hardware Verification and Automated Theorem Proving*, edited by G. Birtwistle and P.A. Subrahmanyam (Springer-Verlag, 1989), pp. 341-386.

3. J. M. Spivey, *The Z Notation*, Prentice Hall International, 1989.

4. J. M. Spivey, *Understanding Z*, Cambridge University Press, 1988.

5. J.M. Spivey, "Free Type Definitions", in *Proceedings of the Third Annual Z Users Meeting*, Oxford University Computing Laboratory, Programming Research group, December 1988.

6. R. D. Arthan, *On Free Type Definitions in Z*, Issue 1.7, ICL Defence Systems, ref DS/FMU/IED/WRK/016, April 1991.

Acknowledgements

The author would like to thank all his colleagues who gave help and advice in writing this report.

Annex

This annex shows how Melham's *define_type* function in HOL (version 1.11) defines the following free type

$$T ::= c \langle\langle A \rangle\rangle \mid d \langle\langle B \times T \times T \rangle\rangle$$

where A and B are given sets, by definitional extension. The annex shows how the representation of the free type, as a set of labelled trees, is performed (part 1), together with the proof of the primitive recursion theorem (PRT) for the free type (part 2). This annex is equivalent to one call of *define_type* for T. The HOL commands are in italics, but the HOL syntax has not been fully adhered to. The syntax that *define_type* expects means that T would actually have to be input as

$$T = c\, A \mid d\, B\, T\, T$$

which means that the constructor d will have type $B \rightarrow T \rightarrow T \rightarrow T$, rather than $(B \times T \times T) \rightarrow T$ as in the Z. Also, it is assumed that A and B already exist before *define_type* is called. This can be achieved by the two commands

```
new_type 0 'A';;
new_type 0 'B';;
```

1. Defining the free type *T*

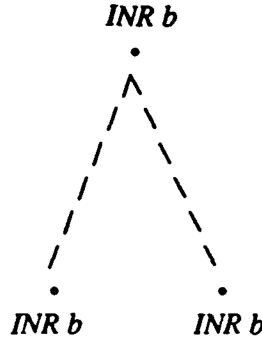
The function *define_type* first defines a predicate *IS_T_REP* below, which is true only of those labelled trees which represent elements of T. The predicate *IS_T_REP* will characterise the subset of *(A + B)ltree* which is to represent the free type T. For any type *, *(*)ltree* is the type of trees, of any shape, labelled with elements of *. The type $A + B$ is the labelled disjoint union of A and B. HOL contains the built-in functions *INL* and *INR* to form elements of $A + B$ from elements of A and B respectively. The tree with top node labelled with v and list of subtrees tl is written in HOL as *Node v tl*. The type of labelled trees has itself been defined using a definitional extension in HOL. The function *LENGTH* gives the length of a list.

```
let IS_T_REP = new_definition('IS_T_REP',
              "IS_T_REP (tree : (A + B)ltree) =
                 TRP
                 (λ v : (A + B)
                   tl : ((A + B)ltree)list •
                        (∃ a : A • v = INL a) ∧ (LENGTH tl = 0)
                      ∨ (∃ b : B • v = INR b) ∧ (LENGTH tl = 2)
                 )
                 tree");;
```

The function *TRP* in *IS_T_REP* is now explained. In *IS_T_REP*, the basic predicate which defines those trees which are representations is

$$(\exists\, a : A \bullet v = INL\ a) \wedge (LENGTH\ tl = 0) \qquad (21)$$
$$\vee\ (\exists\, b : B \bullet v = INR\ b) \wedge (LENGTH\ tl = 2)$$

But unfortunately, this is not quite good enough, since for example, the tree



satisfies (21). But the above tree does not represent any element of the free type *T*. The trouble is, predicate (21) only states what form the top node should have and the *number* of subtrees from the top node. Predicate (21) says nothing about the *form* of the subtrees. To rule out such trees as above, the function *TRP* is required, which basically makes sure that predicate (21) is obeyed all the way down the tree. The function *TRP* is defined in HOL as follows

$$TRP\ \vdash \forall\, P\ v\ tl \bullet TRP\ P\ (Node\ v\ tl) = (P\ v\ tl) \wedge (EVERY\ (TRP\ P)\ tl)$$

where the function *EVERY* is defined as

$$EVERY_DEF\ \vdash (\forall\, P \bullet EVERY\ P\ [] = T) \wedge$$
$$(\forall\, P\ h\ t \bullet EVERY\ P\ (CONS\ h\ t) = (P\ h) \wedge (EVERY\ P\ t)$$

The function *CONS* adds an element to the front of a list. The names *TRP* and *EVERY_DEF* that appear to the left of the two turnstiles, ⊢ above, are simply the names of the definitions, so that they can be used in theorem proving. For example, it must be shown that *IS_T_REP* characterises a non-empty subset of *(A + B)ltree*, that is the following goal must be proved.

$$\exists\, tree : (A + B)ltree \bullet IS_T_REP\ tree$$

An existential witness that can be used for this goal is the tree

INL a
•

for some *a* in *A*. This tree is written in HOL as *Node (INL a)* []. The tactic which proves the goal is

```
EXISTS_TAC "Node (INL a) [] : (A + B)ltree" THEN
REWRITE_TAC [IS_T_REP; TRP; EVERY_DEF] THEN
BETA_TAC THEN
REWRITE_TAC [LENGTH]
```

where *LENGTH* is the definition

$$LENGTH \vdash (LENGTH\ [] = 0) \wedge (\forall h\ t\ .\ LENGTH\ (CONS\ h\ t) = SUC\ (LENGTH\ t))$$

The function *SUC* is the successor function; it adds 1 to its argument. Let *NON_EMPTY* denote the existence theorem just proved.

$$NON_EMPTY \vdash \exists\ tree\ .\ IS_T_REP\ tree$$

The free type *T* is made isomorphic to its representation, that is the subset of *(A + B)ltree* characterised by *IS_T_REP*. Notice that this step requires the theorem just proved.

```
let T_ISO = new_type_definition('T', "IS_T_REP : (A + B)ltree -> bool", NON_EMPTY);;
```

Next, the names *REP_T* and *ABS_T* are given to the isomorphisms. Thus *REP_T* has type $T \rightarrow (A + B)ltree$ and *ABS_T* has type $(A + B)ltree \rightarrow T$.

```
define_new_type_isomorphisms(EXPAND_TY_DEF T_ISO)
```

The function *EXPAND_TY_DEF* above, is built in to HOL. This step also produces some theorems involving *REP_T* and *ABS_T* which are needed in step 2 (proving the PRT), for example

$$\vdash \forall a \bullet ABS_T(REP_T\ a) = a$$
$$\vdash \forall r \bullet IS_T_REP\ r = (REP_T(ABS_T\ r) = r)$$

Finally, the constructors *c* and *d* of *T* are defined.

```
new_definition('c_DEF', "c a = ABS_T(Node (INL a) [])");;
new_definition('d_DEF', "d b t1 t2 = ABS_T(Node (INR b) [REP_T t1; REP_T t2])");;
```

which state what the tree representations of *c a* and $d\ b\ t_1\ t_2$ are.

2. Proving the primitive recursion theorem

The following theorem has been proved in HOL

$$
\begin{array}{l}
TY_DEF_THM \vdash \forall P\ ABS\ REP \bullet \\
\quad antecedents \Rightarrow \\
\quad \forall f \bullet \\
\quad\quad \exists_1 fn \bullet \\
\quad\quad\quad \forall v\ tl \bullet \\
\quad\quad\quad\quad P\ v\ (MAP\ REP\ tl) \Rightarrow \\
\quad\quad\quad\quad fn(ABS(Node\ v\ (MAP\ REP\ tl))) = f(MAP\ fn\ tl)\ v\ tl
\end{array}
$$

This is a general theorem which *define_type* uses to obtain the PRT for T. It can be used to obtain the PRT for any free type defined in terms of labelled trees; it simply has to be instantiated for T, and then simplified. The *antecedents* in the above theorem are theorems obtained from part 1; and can thus soon be removed by modus ponens. So *define_type* instantiates *TY_DEF_THM* for T and then specializes P to be predicate (21) (see part 1), *ABS* to be *ABS_T* and *REP* to be *REP_T*. After modus ponens with the antecedents this produces the new theorem

$$
\begin{array}{l}
\vdash \forall f \bullet \\
\quad \exists_1 fn \bullet \\
\quad\quad \forall v\ tl \bullet \\
\quad\quad\quad (\exists a \bullet v = INL\ a) \wedge (LENGTH(MAP\ REP_T\ tl) = 0) \\
\quad\quad\quad \vee (\exists b \bullet v = INR\ b) \wedge (LENGTH(MAP\ REP_T\ tl) = 2) \\
\quad\quad\quad \Rightarrow \\
\quad\quad\quad fn(ABS_T(Node\ v\ (MAP\ REP_T\ tl))) = f(MAP\ fn\ tl)\ v\ tl
\end{array}
$$

Already it can be seen that this theorem has the basic shape of a PRT. By certain simplifications of the theorem, the details of which are given in [2], the PRT

$$
\begin{array}{l}
\vdash \forall f_1 f_2 \bullet \\
\quad \exists_1 fn \bullet \\
\quad\quad \forall a \bullet fn(c\ a) = f_1\ a \\
\quad\quad \wedge \forall b\ t_1\ t_2 \bullet fn(d\ b\ t_1\ t_2) = f_2\ (fn\ t_1)\ (fn\ t_2)\ b\ t_1\ t_2
\end{array}
$$

is obtained

REPORT DOCUMENTATION PAGE

DRIC Reference Number (if known) ....

Overall security classification of sheet ..................................UNCLASSIFIED..

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the field concerned must be marked to indicate the classification eg (R), (C) or (S).

Originators Reference/Report No.	Month	Year
REPORT 91028	AUGUST	1991

Originators Name and Location
RSRE, St Andrews Road Malvern, Worcs WR14 3PS
Monitoring Agency Name and Location
Title
ON RECURSIVE FREE TYPES IN Z

Report Security Classification	Title Classification (U, R, C or S)
UNCLASSIFIED	U

Foreign Language Title (in the case of translations)
Conference Details

Agency Reference	Contract Number and Period
Project Number	**Other References**

Authors	Pagination and Ref
SMITH, A	37

Abstract

Inconsistent specifications may give rise to false conclusions in reasoning, thus destroying the point of having a specification. This report is concerned with inconsistent specifications which may arise when using the formal specification language Z. In particular, the report is concerned with the inconsistencies that can arise when using recursive free types, and recursive functions defined over recursive free types. The intended audience of the report consists of Z practitioners who wish to avoid writing meaningless specifications.

Abstract Classification (U,R,C or S)
U

Descriptors
Distribution Statement (Enter any limitations on the distribution of the document)
UNLIMITED

980/48