Living with Free Type and Class Union

Jin Song Dong
Software Engineering Group
CSIRO Division of Information Technology
Canberra, ACT 2601, Australia
jsdong@cbr.dit.csiro.au

Abstract

There are two constructs in the formal specification language Object-Z for modelling polymorphic and recursive structures. One construct, the free type, is adopted from the Z specification language. The other, class-union, facilitates polymorphic class declarations. Free type and class-union constructs are respectively based upon the functional value point of view and the object reference point of view. Consequently, the roles these two constructs perform in system modelling are different. In this paper, the free type and class-union constructs are compared and discussed through various examples. The aim of this comparison and discussion is to present guidelines on how to appropriately and effectively use these two constructs to specify polymorphic and recursive structures.

Keywords: formal specification techniques, object-oriented modelling, polymorphic and recursive structures.

1 Introduction

In the formal specification language Z[12], the free type construct is introduced as a convention to represent (polymorphic) union types. Recursive structures can also be modelled by free type definitions. As Object-Z[7, 9, 10] is an object-oriented extension of Z, the free type construct is a legitimate mechanism to capture polymorphic and recursive structures with Object-Z. On the other hand, the class-union[3] construct is also designed to facilitate the specification of polymorphic and recursive structures. Free type and class-union have been developed from different points of view: the free type construct is based more on the notion of functional values while the class-union construct is based more on the notion of object references. This means that the usages of these two constructs in system modelling are different and it is therefore important to study the differences between these two constructs so that the appropriate one can be chosen when specifying systems. This paper presents a comparison of the free type and the class-union constructs and provides guidelines for applying them in system modelling.

Section 2 outlines the definition of the free type and the class-union constructs. Section 3 presents a case study specifying a currency exchange office which contains bank notes in different currencies. In the case study, the free type and the class-union constructs are applied separately to model the polymorphic type of a bank note, and the resultant representations are compared. Extendibility of the representations is also discussed. Continuing with the discussion on extendibility, Section 4 demonstrates that free type and class union can be combined to improve the extendibility of the specification of a communication channel. Section 5 presents a case study specifying a recursive binary tree structure, with an operation to insert a node to a non-empty tree. Using the two constructs to give different specifications of the binary tree further clarifies the distinctions between the functional value point of view (free type) and the object reference point of view (class-union). Suitability of the two constructs for modelling other recursive structures is also discussed. Finally, in Section 6, the consistency issue of free type definitions [1, 11] is extended to class-union.

2 Background to Free Type and Class Union

In this section, we outline the definition of the free type and the class-union constructs. (It is assumed readers have a basic understanding of Z and Object-Z.)

2.1 Free Types

The Z free type construct[12] is a way of combining two defined sets into a new set. As Z is based on typed set theory, it is invalid to write:

$$Value == \mathbb{B} \cup \mathbb{Z} \cup Char$$

where $\mathbb{B}$, $\mathbb{Z}$ and $Char$ denote booleans, integers and characters respectively. In Z, a more elaborate approach (free type definition) is used to model the above situation:

$$Value ::= bool\langle\langle \mathbb{B} \rangle\rangle \mid int\langle\langle \mathbb{Z} \rangle\rangle \mid char\langle\langle Char \rangle\rangle$$

In the above free type definition, the *bool*, *int* and *char* constructors are total injections (total one-to-one functions) from the sets $\mathbb{B}$, $\mathbb{Z}$ and $Char$ to $Value$.

A consequence of using free types is that disjoint sets can be labeled by the names of constructors and grouped together. For instance, an element, '3', of $\mathbb{Z}$ (integers) corresponds to an element, '$int(3)$', of *Value*; an element of *Value*, say '*val*' which is in the range of *int*, corresponds to an element of $\mathbb{Z}$, '$int^{\sim}(val)$' (where the super-script $^{\sim}$ denotes function inverse).

Free type definitions can be recursive. For example, a simple linked-list can be specified in Z by the free type definition

$$LinkedList ::= nil \mid next\langle\langle \mathbb{Z} \times LinkedList \rangle\rangle$$

This introduces a new type *LinkedList*, a constant *nil* of type *LinkedList* and an injective constructor function *next*, that, given an integer and a linked-list, returns a linked-list. For example, elements '*nil*', '$next(3, nil)$', '$next(1, next(3, nil))$' etc are members of *LinkedList*.

A free type definition can be transferred into other more primitive Z constructs, namely given types and a set of constructor functions (axiomatic descriptions). For instance, the free type *LinkedList* can be transferred into the following definitions:

$$[LinkedList]$$

$$\begin{array}{|l}
nil : LinkedList \\
next : (\mathbb{Z} \times LinkedList) \qquad LinkedList \\
\hline
disjoint\langle \{nil\}, \operatorname{ran} next \rangle \\
\forall W : \mathbb{P}\, LinkedList \bullet \\
\qquad \{nil\} \cup next(\!|\ \mathbb{Z} \times W\ |\!) \subseteq W \\
\qquad\qquad \Rightarrow LinkedList \subseteq W
\end{array}$$

In the above axiomatic definition, the first predicate constrains the range of *next* so that it does not contain the constant *nil*. The second predicate constrains the set *LinkedList* to be closed under the constructor *next*. For a detailed discussion of free types, see [1, 11, 12].

2.2 Class Union

The following is a brief explanation of the class-union construct which is detailed in [2, 3].

Underlying the semantics of an Object-Z specification is the understanding that each object has a unique persistent identity that distinguishes it from every other object, regardless of class[8]. An object's identity is invariant irrespective of changes to its state.

If P is the name of a class, let the identifier P also denote the set of identities of possible objects of class P. In any system being specified, $\mathbb{O}$ denotes the universe of all object identities, regardless of class, that may be in the system.

If P and Q are distinct classes in the system,

$$P \subseteq \mathbb{O}, \ \ Q \subseteq \mathbb{O} \text{ and } P \cap Q = \varnothing.$$

Informally, we do not distinguish between an object and its identity, i.e. if we refer to some *object* in $\mathbb{O}$, it is with the understanding that we are in fact referring to the object whose identity is in $\mathbb{O}$. As object identities uniquely identify objects, no ambiguity arises. Subsets of $\mathbb{O}$ can be formed by the union of sets of object identities, e.g. if P and Q are distinct classes, the class-union $P \cup Q$ denotes a collection of object identities which are the union of object identities of P and the object identities of Q. In this case, if $R \mathrel{\widehat=} P \cup Q$ then R is not itself a class but only a set of object identities.

2.2.1 Polymorphic Core

Associated with any object is a set of attributes and a set of operations. Given a collection of objects, possibly from different classes, an attribute named *attr* of type T, say, is said to be *polymorphic with respect to the collection* if every object in the collection has an associated attribute named *attr* of type T. An operation named *Op*, say, is said to be *polymorphic with respect to the collection* if

- every object in the collection has an associated operation named *Op*;
- for each object in the collection, the parameter list of the operation *Op* is the same, i.e. the input and output parameters are identically named and typed.

Associated with any collection of objects is the set of all attributes and operations that are polymorphic with respect to the collection. This set of attributes and operations will be called the *polymorphic core* of the collection. For instance, if

$$C \mathrel{\widehat=} A \cup B$$

where A and B are the following classes

$$\begin{array}{|l}
\multicolumn{1}{l}{A} \\
\hline
\begin{array}{|l} a, c, x : \mathbb{Z} \end{array} \\
\\
\begin{array}{|l} \multicolumn{1}{l}{Change} \\ \hline \Delta(a) \\ i? : \mathbb{Z} \\ \hline a' = x + i? \end{array} \\
\\
\begin{array}{|l} \multicolumn{1}{l}{ChangeC} \\ \hline \Delta(c) \\ i? : \mathbb{Z} \\ \hline c' = x + i? \end{array}
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{B} \\
\hline
\begin{array}{|l} b, d, x : \mathbb{Z} \end{array} \\
\\
\begin{array}{|l} \multicolumn{1}{l}{Change} \\ \hline \Delta(b) \\ i? : \mathbb{Z} \\ \hline b' = x * i? \end{array} \\
\\
\begin{array}{|l} \multicolumn{1}{l}{ChangeD} \\ \hline \Delta(d) \\ i? : \mathbb{Z} \\ \hline d' = x * i? \end{array}
\end{array}$$

then the polymorphic core of C is the attribute x and the operation *Change*.

2.2.2 Typing

Given a declaration $s : C$, the expressions '$s.x$' and '$s.Change$' can be interpreted polymorphically regardless of the actual class of s; consequently, the expressions '$s.x$' and '$s.Change$' are well typed under the type scope '$s : C$'. In contrast, the expressions '$s.a$' and '$s.ChangeC$' conflict with the situation when the actual class of s is B because 'a' and '$ChangeC$' are features in A but not in B. Therefore, those expressions '$s.a$' and '$s.ChangeC$' are considered as ill-typed (under the type scope '$s : C$') unless the type of s is narrowed ('$\succ\!\!\succ$') to the subtype — class A, i.e. the expressions '$s \succ\!\!\succ A \bullet s.a$' and '$[s \succ\!\!\succ A] \bullet s.ChangeC$' are well typed.

In general, suppose a reference e is originally declared as '$e : E$' (where E is a class-union), then the expressions '$e \succ\!\!\succ F \bullet p(e.attr)$' and '$[e \succ\!\!\succ F] \bullet e.op$' are well typed (or valid) if

- F is a class in E or is a class-union of classes which are included in E, i.e. $F \subseteq E$, and
- the attribute $attr$ and the operation op are in the polymorphic core of F.

The expressions '$e \succ\!\!\succ F \bullet p(e.attr)$' and '$[e \succ\!\!\succ F] \bullet e.op$' are logically equivalent to '$\forall e_0 : F \bullet e_0 = e \Rightarrow p(e_0.attr)$' and '$[e_0 : F \mid e_0 = e] \bullet e_0.op$' respectively.

Notice that if $F \subseteq E$, then the polymorphic core of F is (often) larger than or (at least) equal to the polymorphic core of E.

Type narrowing facilitates the type checking for polymorphic references in Object-Z. It is reminiscent of the downcasting in C++.

In the following sections, the free type and the class-union constructs are applied to specify polymorphic and recursive structures and their implications are explored.

3 Polymorphism

Consider a situation where a currency exchange office (CEO) consists of a collection of franc and dollar bank notes. Suppose an operation can be performed in a CEO to output the total value of the bank notes in dollars. Although the two currencies can be both modelled as the same mathematical kind ($\mathbb{N}$), it is necessary to label each note to distinguish the two different kinds of currency.

The general type of bank note is a polymorphic type which can be modelled as a free type or as a class-union.

Firstly, consider the general type of a bank note specified by a free type.

3.1 Modelling Bank Notes in Free Type

If a bank note is modelled as

$$Note_{ft} ::= franc\langle\!\langle \mathbb{N} \rangle\!\rangle \mid dollar\langle\!\langle \mathbb{N} \rangle\!\rangle$$

(where the sub-script '$_{ft}$' indicates that the term is defined by a free type) then two notes of one currency having the same value, e.g. two ten dollar notes, will be indistinguishable. Therefore bank note identity needs to be modelled in the definition.

Let $[FId]$ and $[DId]$ represent the identities of franc and dollar bank notes (like the serial numbers of the bank notes) respectively. The two currencies can be modelled as

$$\begin{array}{|l} Francs : FId \twoheadrightarrow \mathbb{N} \\ Dollars : DId \twoheadrightarrow \mathbb{N} \end{array}$$

where the ranges of the functions represent the face value of the bank notes. Then, the general type of a bank note can be represented as:

$$BankNote_{ft} ::= franc\langle\!\langle Francs \rangle\!\rangle \mid dollar\langle\!\langle Dollars \rangle\!\rangle$$

Suppose the exchange rate from francs to dollars is

$$\begin{array}{|l} \mathcal{F}to\mathcal{D} : \mathbb{R} \end{array}$$

(We assume that all arithmetic operators which are defined for $\mathbb{N}$ are also valid for $\mathbb{R}$ and $\mathbb{N}$ is a subtype of $\mathbb{R}$.)

A CEO can be modelled as

CEO_{ft}

$$notes : \mathbb{P}\, BankNote_{ft}$$

$TotalInDollars$

$$total! : \mathbb{R}$$

$$\begin{aligned} total! = {} & \textstyle\sum n : (notes \cap \operatorname{ran} dollar) \bullet \\ & \quad second(dollar^{\sim}(n)) \\ & + \\ & \textstyle\sum n : (notes \cap \operatorname{ran} franc) \bullet \\ & \quad second(franc^{\sim}(n)) * \mathcal{F}to\mathcal{D} \end{aligned}$$

(The function $second$ denotes the second element of an ordered pair.)

Consider now the use of the class-union construct to model the bank notes.

3.2 Modelling Bank Notes in Class Union

If the two kinds of bank notes are modelled as objects of two different classes, $Franc$ and $Dollar$, then a general type of a bank note can be defined as a class-union:

$$BankNote \mathrel{\widehat{=}} Franc \cup Dollar$$

where $Franc$ and $Dollar$ are specified as:

$Franc$

$$\begin{array}{l} face\_value : \mathbb{N} \\ \Delta \\ dlr : \mathbb{R} \\ \hline dlr = face\_value * \mathcal{F}to\mathcal{D} \end{array}$$

$Dollar$

$$\begin{array}{l} face\_value : \mathbb{N} \\ \Delta \\ dlr : \mathbb{R} \\ \hline dlr = face\_value \end{array}$$

Note that attributes under the Δ symbol in the state schema of a class are secondary attributes. Although the secondary attributes add no further information to the specification in that they ultimately depend on primary attributes (those above the Δ symbol), they can add clarity to a specification. For a detailed discussion on secondary attributes see [6, 8].

In the classes *Franc* and *Dollar*, the secondary attribute *dlr* (dollar) is used to represent the value of a bank note in dollars. Now the specification of the CEO can be represented as

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{CEO}} \\
\hline
\begin{array}{|l}
\hline
notes : \mathbb{P}\, BankNote \\
\hline
\end{array} \\
\\
\begin{array}{|l}
\multicolumn{1}{l}{\textit{TotalInDollars}} \\
\hline
total! : \mathbb{R} \\
\hline
total! = \sum n : notes \bullet n.dlr \\
\hline
\end{array} \\
\hline
\end{array}
$$

Comparing these two models, CEO_{ft} and *CEO*, we see that one of the differences is that the identity (serial number) of a bank note is implicitly modelled by the semantics of the classes *Franc* and *Dollar* in *CEO* (a class denotes a set of object identities of the class), while in CEO_{ft}, the serial number of each bank note must be explicitly specified by given types *FId* and *DId*. Another observation is that the definition of the operation *TotalInDollars* in *CEO* is much simpler than that in CEO_{ft}. This is because the attribute *dlr* is in the polymorphic core of the class-union *BankNote* so that the term '$n.dlr$' in the operation *TotalInDollars* of *CEO* is interpreted polymorphically regardless of the class of n. In contrast, the operation *TotalInDollars* of CEO_{ft} must be specified using case analysis.

3.3 Extendibility

If we extend the specification to include Yen bank notes in the currency exchange office with the exchange rate from yen to dollars as

$$| \; \mathcal{Y}to\mathcal{D} : \mathbb{R},$$

then in the free type representation we need to

- introduce

  $$| \quad Yen : YId \rightarrowtail \mathbb{N}$$

  where *YId* represents all the serial numbers of Yen notes;

- extend the free type definition $BankNote_{ft}$ to

  $$\begin{array}{rl} BankNote_{ft} \; ::= & franc\langle\langle Francs\rangle\rangle \mid \\ & dollar\langle\langle Dollars\rangle\rangle \mid \\ & yen\langle\langle Yen\rangle\rangle; \end{array}$$

- and modify the predicate part of the operation *TotalInDollars* as

  $$\begin{array}{rl} Total! = & \sum n : (notes \cap \operatorname{ran} dollar) \bullet \\ & \qquad second(dollar^{\sim}(n)) \\ & + \\ & \sum n : (notes \cap \operatorname{ran} franc) \bullet \\ & \qquad second(franc^{\sim}(n)) * \mathcal{F}to\mathcal{D} \\ & + \\ & \sum n : (notes \cap \operatorname{ran} yen) \bullet \\ & \qquad second(yen^{\sim}(n)) * \mathcal{Y}to\mathcal{D} \end{array}$$

However in the class-union representation, we need only

- add the class:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Yen}} \\
\hline
\begin{array}{|l}
\hline
face\_value : \mathbb{N} \\
\Delta \\
dlr : \mathbb{R} \\
\hline
dlr = face\_value * \mathcal{Y}to\mathcal{D} \\
\hline
\end{array} \\
\hline
\end{array}
$$

- and extend the class-union *BankNote* to

  $$BankNote \mathrel{\widehat{=}} Franc \cup Dollar \cup Yen$$

The specification of the class *CEO* remains unchanged. The class-union representation of a CEO is more extendible because the features in the polymorphic core of a class-union can be applied polymorphically; in contrast, there is no notion of polymorphic core in a free type definition. This example clearly favours the class-union representation. However, this does not suggest the complete replacement of the free type construct by the class-union construct in Object-Z specifications. In the following section, an example demonstrates that a free type can be used to construct the polymorphic core of a class-union in a specification so that the combination of the two improves the extendibility of the specification.

4 Construction of a Polymorphic Core

One kind of polymorphism supported by essentially all programming languages is function and operator overloading. For example, in procedural languages the assignment statement is a polymorphic operator because it supports both the assignment of the value of a boolean expression to a boolean variable, and the assignment of the value of an number expression to a number variable. Based on this idea, in [2], a specification of a polymorphic channel was presented. In this paper, we now extend the channel specification to allow a string of characters (in addition to natural numbers and boolean values) to be passed through the channel. The objects which are allowed to be passed through the channel are modelled as a class-union:

$$Object \mathrel{\widehat{=}} Nat \cup Bool \cup String$$

where classes *Nat*, *Bool* and *String* are defined as

Nat

$$n : \mathbb{N}$$

INIT: $n = 0$

Update
$$\Delta(n) \quad x? : \mathbb{N}$$
$$\overline{\quad} \quad n' = n + x?$$

OutVal
$$x! : \mathbb{N}$$
$$\overline{\quad} \quad x! = n$$

Bool

$$b : \mathbb{B}$$

INIT: $b = true$

Update
$$\Delta(b) \quad x? : \mathbb{B}$$
$$\overline{\quad} \quad b' = x?$$

OutVal
$$x! : \mathbb{B}$$
$$\overline{\quad} \quad x! = b$$

String

$$s : \mathrm{seq}\ Char$$

INIT: $s = \langle\, \rangle$

Update
$$\Delta(s) \quad x? : \mathrm{seq}\ Char$$
$$\overline{\quad} \quad s' = s \quad x?$$

OutVal
$$x! : \mathrm{seq}\ Char$$
$$\overline{\quad} \quad x! = s$$

The class *Channel* is specified as

Channel

$$sender, receiver : Object$$
$$\overline{\quad} \quad sender \neq receiver$$

Switch
$$\Delta(sender, receiver) \quad s?, r? : Object$$
$$\overline{\quad} \quad sender' = s? \wedge receiver' = r?$$

$$\begin{aligned} Trans \mathrel{\widehat=} {} & [sender, receiver \succ Nat] \bullet \\ & \quad sender.OutVal \parallel receiver.Update \\ & [\!] \\ & [sender, receiver \succ Bool] \bullet \\ & \quad sender.OutVal \parallel receiver.Update \\ & [\!] \\ & [sender, receiver \succ String] \bullet \\ & \quad sender.OutVal \parallel receiver.Update \end{aligned}$$

Operation *Switch* enables the *sender* and the *receiver* to be changed.

Operation *Trans* is defined using the choice operator ‘ $[\!]$ ’ (see [8]). The pre-conditions for each of the alternatives of *Insert* are disjoint, so that at most one alternative is invoked. A kind of polymorphism is represented by the *Trans* operation: this operation can be performed on the objects *sender* and *receiver* (providing they are the same class) regardless of whether that class is *Nat*, *Bool* or *String*. However, the representation of the operation *Trans* above is obviously awkward. The reason for the repetitious definition of *Trans* is that the component operations *OutVal* and *Update* cannot be applied polymorphically since they are not in the polymorphic core of the class-union *Object* (the parameter types of the operations' input-output interfaces are different). However, by using a free type to represent the general value type of $\mathbb{N}$, $\mathbb{B}$ and seq *Char*, i.e.

$$Val ::= nat\langle\!\langle \mathbb{N} \rangle\!\rangle \mid bool\langle\!\langle \mathbb{B} \rangle\!\rangle \mid str\langle\!\langle \mathrm{seq}\ Char \rangle\!\rangle$$

and re-specifying *Nat*, *Bool* and *String* as

Nat

$$n : Val$$
$$\overline{\quad} \quad n \in \mathrm{ran}\ nat$$

INIT: $n = nat(0)$

Update
$$\Delta(n) \quad x? : Val$$
$$\overline{\quad} \quad x? \in \mathrm{ran}\ nat$$
$$n' = nat(nat^{\sim}(n) + nat^{\sim}(x?))$$

OutVal
$$x! : Val$$
$$\overline{\quad} \quad x! = n$$

Bool

$$b : Val$$
$$\overline{\quad} \quad b \in \mathrm{ran}\ bool$$

INIT: $n = bool(true)$

Update
$$\Delta(b) \quad x? : Val$$
$$\overline{\quad} \quad x? \in \mathrm{ran}\ bool$$
$$b' = x?$$

OutVal
$$x! : Val$$
$$\overline{\quad} \quad x! = b$$

String

$$s : Val$$
$$\overline{\quad} \quad s \in \mathrm{ran}\ str$$

INIT: $n = str(\langle\, \rangle)$

Update
$$\Delta(s) \quad x? : Val$$
$$\overline{\quad} \quad x? \in \mathrm{ran}\ str$$
$$s' = str(str^{\sim}(s) \quad str^{\sim}(x?))$$

OutVal
$$x! : Val$$
$$\overline{\quad} \quad x! = s$$

the operations *Update* and *OutVal* now belong to the polymorphic core of the class-union *Object*. The *Trans* operation in the class *Channel* can now be specified polymorphically as

$$Trans \mathrel{\widehat{=}} sender.OutVal \parallel receiver.Update.$$

Note that the precondition of the operation *Update* in each class *Nat*, *Bool* and *String* ensures that the operation succeeds only if *sender* and *receiver* belong to the same class.

Furthermore, if *Object* is extended to include more new kinds of objects, the operation *Trans* remains unchanged as long as the polymorphic core of the original class-union *Object* is preserved, i.e. the use of free type ensures the specification of the channel is easily extendible.

In the next section, we investigate the role of applying free types and class-unions to model recursive structures.

5 Recursive Structures

Consider an ordered binary tree of integers with an operation *Insert* to insert a node into an existing tree. The general structure of the tree can be modelled using a free type definition

$$\begin{array}{lll} TrNode_{ft} & ::= & nilnode \mid \\ & & valnode\langle\!\langle TrNode_{ft} \times \mathbb{Z} \times TrNode_{ft} \rangle\!\rangle \end{array}$$

Based on this free type structure, ordered binary trees can be modelled as

$$\begin{array}{|l} OrdTrNode : \mathbb{P}\, TrNode_{ft} \\ \hline \forall\, t : TrNode_{ft} \bullet \\ \quad t \in OrdTrNode \Leftrightarrow \\ \quad\quad t = nilnode \lor \\ \quad\quad\quad \exists\, i : \mathbb{Z};\ l, r : TrNode_{ft} \bullet \\ \quad\quad\quad\quad t = valnode(l, i, r) \\ \quad\quad\quad\quad \{l, r\} \subseteq OrdTrNode \\ \quad\quad\quad\quad \forall\, n : nums(l) \bullet n < i \\ \quad\quad\quad\quad \forall\, n : nums(r) \bullet n \geq i \end{array}$$

where *nums* is an auxiliary function which takes a tree and returns a set of numbers in the tree:

$$\begin{array}{|l} nums : TrNode_{ft} \rightarrow \mathbb{P}\,\mathbb{Z} \\ \hline \forall\, t : TrNode_{ft} \bullet \\ \quad t = nilnode \Rightarrow nums(t) = \varnothing \\ \quad \exists\, i : \mathbb{Z};\ l, r : TrNode_{ft} \bullet \\ \quad\quad t = valnode(l, i, r) \Rightarrow \\ \quad\quad\quad nums(t) = nums(l) \cup \{i\} \cup nums(r) \end{array}$$

The *Insert* operation can be modelled as a function which takes a tree node and a given root tree and returns a new tree:

$$\begin{array}{|l} Insert_{ft} : OrdTrNode \times OrdTrNode \nrightarrow OrdTrNode \\ \hline \mathrm{dom}\, Insert_{ft} = \{(n, t) : OrdTrNode \times OrdTrNode \\ \quad\quad \mid \exists\, i : \mathbb{Z} \bullet n = valnode(nilnode, i, nilnode)\} \\ \forall (n, root) : \mathrm{dom}\, Insert_{ft} \bullet \\ \quad root = nilnode \Rightarrow Insert_{ft}(n, root) = n \\ \quad \exists\, j : \mathbb{Z};\ l, r : OrdTrNode \bullet \\ \quad\quad root = valnode(l, j, r) \Rightarrow \\ \quad\quad\quad (i > j \Rightarrow Insert_{ft}(n, root) = \\ \quad\quad\quad\quad\quad valnode(l, j, Insert_{ft}(n, r)) \\ \quad\quad\quad\ i \leq j \Rightarrow Insert_{ft}(n, root) = \\ \quad\quad\quad\quad\quad valnode(Insert_{ft}(n, l), j, r)) \end{array}$$

Now we consider the representation of the binary tree using the class-union construct. If the binary tree nodes are viewed as objects, they can be modelled as a class-union:

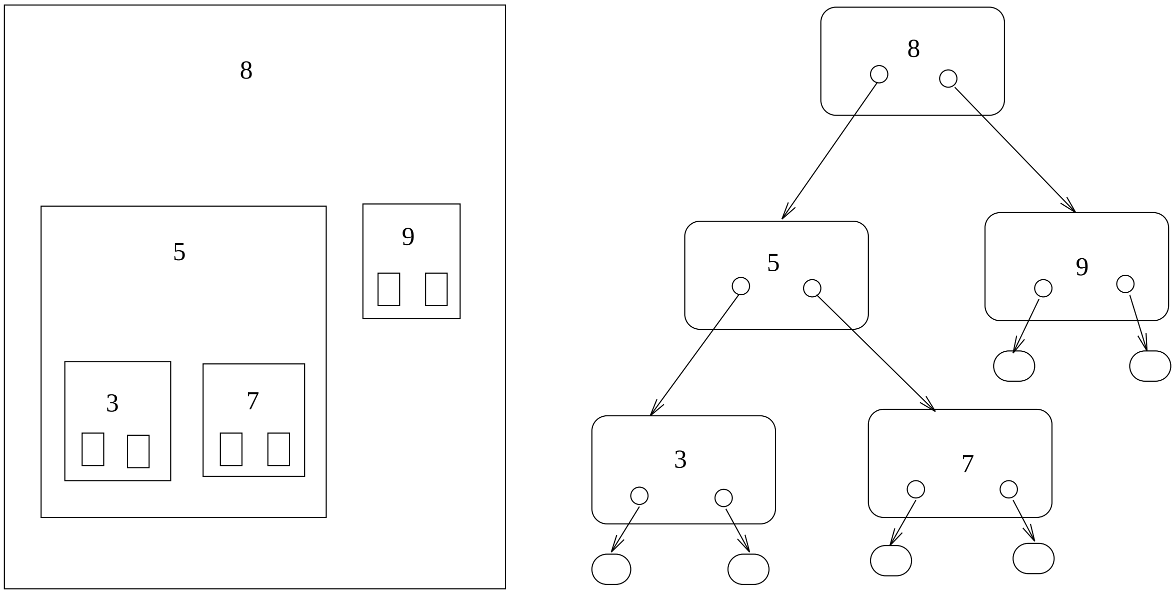
$$TrNode \mathrel{\widehat{=}} NullNode \cup ValNode$$

where the *NullNode* class represents the *nilnode* objects

$$\begin{array}{|l} \hline NullNode \\ \hline \\ \hline \end{array}$$

and the *ValNode* class represents non-nil tree nodes

$$\begin{array}{|l} \textit{ValNode} \\ \hline \begin{array}{|l} val : \mathbb{Z} \\ l, r : TrNode_{\copyright} \\ \hline l \neq r \end{array} \qquad \begin{array}{|l} \textsc{Init} \\ \hline \{l, r\} \subseteq NullNode \end{array} \\ \\ \begin{array}{|l} ToLeft \\ \hline n? : ValNode \\ \hline n?.\textsc{Init} \\ n?.val < val \end{array} \qquad \begin{array}{|l} ToRight \\ \hline n? : ValNode \\ \hline n?.\textsc{Init} \\ n?.val \geq val \end{array} \\ \\ \begin{array}{|l} ChangeL \\ \hline \Delta(l) \\ n? : ValNode \\ \hline l \in NullNode \\ l' = n? \end{array} \qquad \begin{array}{|l} ChangeR \\ \hline \Delta(r) \\ n? : ValNode \\ \hline r \in NullNode \\ r' = n? \end{array} \\ \\ Insert \mathrel{\widehat{=}}\ \ ToLeft \bullet \\ \quad\quad\quad\quad\quad\quad ChangeL \\ \quad\quad\quad\quad\quad\quad [\!] \\ \quad\quad\quad\quad\quad\quad [l \succ\!\!\!- ValNode] \bullet l.Insert \\ \quad\quad\quad [\!] \\ \quad\quad\quad ToRight \bullet \\ \quad\quad\quad\quad\quad\quad ChangeR \\ \quad\quad\quad\quad\quad\quad [\!] \\ \quad\quad\quad\quad\quad\quad [r \succ\!\!\!- ValNode] \bullet r.Insert \\ \hline \end{array}$$



free type representation   class-union representation

Figure 1: Representations of a binary tree.

The containment notation '©' ensures the tree object reference structure of a *ValNode* object. The general notion of object containment '©' ensures that an object cannot directly or indirectly contain itself; and an object cannot be directly contained within two distinct objects. (For a detailed discussion of object containment, see [4].)

The pre-condition for the *Insert* operation and the state of *INIT* ensures the order property of a binary tree. Now consider the following two specifications of a non-empty tree object with an operation to insert a node at the root of the tree (one is specified by using the free type representation of a tree node and the other is specified by using the class-union):

$Tree_{ft}$

- $root : OrdTrNode$

- $\textsc{Init}$
  - $\exists\, i : \mathbb{Z} \bullet$
    $root = valnode(nilnode, i, nilnode)$

- $Insert$
  - $\Delta(root)$
  - $n? : OrdTrNode$
  - ---
  - $(n?, root) \in \mathrm{dom}\, Insert_{ft}$
  - $root' = Insert_{ft}(n?, root)$

$Tree$

- $root : ValNode$

- $\textsc{Init}$
  - $root.\textsc{Init}$

- $Insert \mathrel{\widehat{=}} [n? : ValNode] \bullet root.Insert$

In $Tree_{ft}$ (using free type), the value of *root* is changed whenever the operation *Insert* is applied; in contrast, in *Tree* (using class-union), only a particular component tree node of the root tree *root* changes its value when the operation *Insert* is applied; the value of *root* is unchanged because the value of *root* is the identity of the root node of the tree.

The two representations of a binary tree in free type and in class-union are very different in style. This difference is due to the different views (value compared with object reference) which free type and class union are based on. Figure 1 captures these different views of a binary tree pictorially.

In this example, it is debatable which representation (by using free type or class union) is more suitable for describing a binary tree. The preference depends on how one views the context.

However, in general, the class-union representation is more suitable for modelling recursive structures because the reference semantics of an Object-Z class supports cross referencing (object sharing). For instance, a doubly linked list node (of Figure 2) can be specified in class-union as

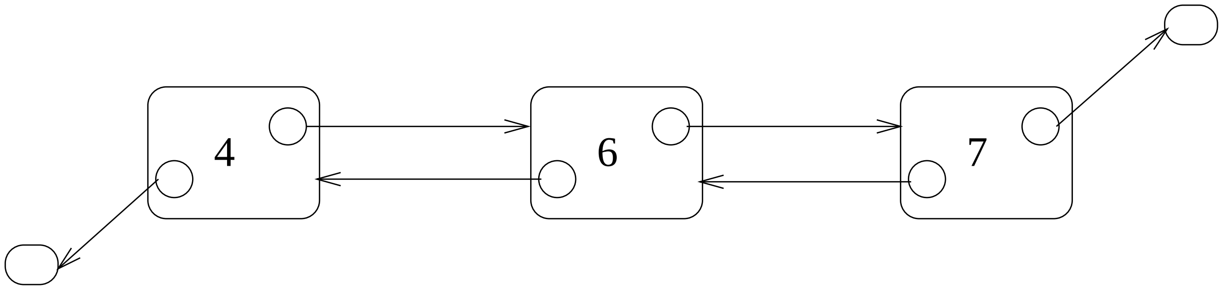


Figure 2: A doubly linked list.

$DLLNode \mathrel{\widehat{=}} NullNode \cup DLLValueNode$

where $DLLValueNode$ is specified as:

$DLLValueNode$

$val : \mathbb{Z}$
$next, prev : DLLNode$

$next \succ DLLValueNode \bullet next.prev = self$
$prev \succ DLLValueNode \bullet prev.next = self$

INIT

$\{next, prev\} \subseteq NullNode$

[details of operations omitted]

On the other hand, it is impossible to represent the doubly linked list structure using free types because the components of a free type are partitioned; sharing between them is not possible.

6 Consistency of Definitions

If it is not possible to find a model which satisfies a specification, then the specification is considered to be inconsistent. In general, it is undecidable whether or not a Z or Object-Z specification is consistent. However, discussion of this issue can assist specifiers to avoid writing meaningless specifications. Based on the work on consistency of recursive free types by Arthan[1] and Smith[11], we discuss the consistency issue of the class-union construct.

Consider the following free type definition:

$$X_{ft} ::= set\langle\langle \mathbb{P}\, X_{ft} \rangle\rangle \mid int\langle\langle \mathbb{Z} \rangle\rangle$$

There is no total injective function from $\mathbb{P}\, X_{ft}$ to X_{ft} because, however large X_{ft} is, there are many more members of $\mathbb{P}\, X_{ft}$ than there are members of X_{ft}. Therefore a model for the free type X_{ft} cannot exist. (The detail of the proof can be found in [1, 11].)

The class-union representation of X_{ft} is:

$$X \mathrel{\widehat{=}} Set \cup Int$$

where Set and Int are:

Set

$s : \mathbb{P}\, X$

Int

$i : \mathbb{Z}$

As a class-union denotes a set of object identities rather than a set of possible object state values, the value of the attribute s in Set is a subset of object identities in X. Furthermore, self, cross and cyclic object referencing is allowed in class definitions; there are no restrictions on what value s can take. Therefore the representation for X is consistent. If the state attribute s of Set is specified as $s : \mathbb{P}\, X_{©}$ so that s represents the contained objects (which is more close to the intended structure specified by X_{ft}), then the value of s is restricted by the implicit state invariant imposed by the notion of object containment. However, the specification of X is still consistent as X essentially models a forest.

Now let's consider another free type definition.

$$Y_{ft} ::= fun\langle\langle Y_{ft} \rightarrow \mathbb{Z} \rangle\rangle \mid int\langle\langle \mathbb{Z} \rangle\rangle$$

Again, the free type Y_{ft} does not exist as there is also no total injective function from $Y_{ft} \rightarrow \mathbb{Z}$ to Y_{ft}.

The corresponding class-union representation of Y_{ft} is

$$Y \mathrel{\widehat{=}} Fun \cup Int$$

where Fun and Int are:

Fun

$f : Y \rightarrow \mathbb{Z}$

Int

$i : \mathbb{Z}$

This representation for Y is also consistent. If the state attribute f in the class Fun is modelled as $f : Y_{©} \rightarrow \mathbb{Z}$, then Y becomes inconsistent. This is because every object in Y paired with a natural number is contained in an object of Fun, and $Fun \subseteq Y$ (as $Y \mathrel{\widehat{=}} Fun \cup Int$); therefore, an object of Fun must contain itself, and this contradicts the cyclic property of containment (i.e. an object cannot directly or indirectly contain itself[4]). Hence no Fun object exists and the specification is inconsistent. However this inconsistency is caused by the conflict between the constraint of the total function and the property of object containment rather than the notion of class-union.

7 Conclusion

In this paper, free type and class-union constructs have been applied to specify a number of polymorphic and recursive systems. By comparing these specifications, our conclusions are:

- The notion of the polymorphic core of a class-union supports the extendibility of system specifications. In contrast, there is no such notion in a free type. (Whether the notion of a class-union's polymorphic core can be extended to cover free types remains to be further investigated.)

- However, a free type definition can contribute to the construction of the polymorphic core of a class-union so that a combination of the two can improve the extendibility of a system specification. This idea has been successfully applied to specify the denotational semantics of programming languages in Object-Z[2, 5].

- Free type and class-union are based on different views. These different views are particularly evident when recursive structures are modelled. Furthermore, the value view point of free types cannot be applied to modelling recursive structures which involve cross-referencing (sharing).

- The inconsistency cases in free type definitions essentially disappear in class-union structures.

As was illustrated in the binary tree example, it is in general debatable which representation, free type or class-union, is more suitable for modelling a system. However, if the identity of a polymorphic entity is as important as the content of the entity (e.g. as in the bank note example), then in general class-union is the better choice.

It is hoped this paper provides useful guidelines on how to model polymorphic and recursive structures by choosing the appropriate constructs in Object-Z specifications.

Acknowledgements

The author would like to thank Roger Duke for his detailed comments on earlier drafts of this paper. The author would also like to thank Steven Atkinson, Andrew Hussey, Gordon Rose and the anonymous referees for many helpful suggestions.

References

[1] R. Arthan. On free type definitions in Z. In *Proceedings of the Sixth Annual Z-User Meeting*, pages 40–58, University of York, Dec 1991.

[2] J. S. Dong. *Formal Object Modelling Techniques and Denotational Semantics Studies.* PhD thesis, The University of Queensland, (forthcoming) 1995.

[3] J. S. Dong and R. Duke. Class Union and Polymorphism. In C. Mingins, W. Haebich, J. Potter, and B. Meyer, editors, *Proc. 12th International Conference on Technology of Object-Oriented Languages and Systems. TOOLS 12 & 9*, pages 181–190. Prentice-Hall, November 1993.

[4] J. S. Dong and R. Duke. The Geometry of Object Containment. *Object-Oriented Systems*, 2(1):41–63, Chapman & Hall, March 1995.

[5] J. S. Dong, R. Duke, and G. Rose. An Object-Oriented Approach to the Semantics of Programming Languages. In G. Gupta, editor, *Proc. 17th Annual Computer Science Conference (ACSC'17)*, pages 767–775, NZ, January 1994.

[6] J. S. Dong, G. Rose, and R. Duke. The Role of Secondary Attributes in Formal Object Modelling. To appear in *The IEEE International Conference on Engineering Complex Computer Systems (ICECCS'95)* Florida, November 1995.

[7] R. Duke, P. King, G. Rose, and G. Smith. The Object-Z specification language. In T. Korson, V. Vaishnavi, and B. Meyer, editors, *Technology of Object-Oriented Languages and Systems: TOOLS 5*, pages 465–483. Prentice-Hall, 1991.

[8] R. Duke and G. Rose. Modelling object identity. In *Proc. 16th Australian Comput. Sci. Conf. (ACSC-16)*, pages 93–100, February 1993.

[9] R. Duke, G. Rose, and G. Smith. Object-Z: a Specification Language Advocated for the Description of Standards. To appear in a special issue of *Computer Standards and Interfaces* on Formal Methods and Standards, September 1995.

[10] G. Rose. Object-Z. In S. Stepney, R. Barden, and D. Cooper, editors, *Object Orientation in Z*, Workshops in Computing, pages 59–77. Springer-Verlag, 1992.

[11] A. Smith. On recursive free types in Z. In *Proceedings of the Sixth Annual Z-User Meeting*, pages 3–39, University of York, December 1991.

[12] J.M. Spivey. *The Z Notation: A Reference Manual.* International Series in Computer Science. Prentice-Hall, 2nd edition, 1992.