Object specifications: Larch meets dependencies

K. Rustan M. Leino and Raymie Stata
9 November 1995

Digital's Systems Research Center
130 Lytton Ave., Palo Alto, CA 94301, U.S.A.
rustan@pa.dec.com, raymie@larch.lcs.mit.edu

We take on the challenge of writing a specification for $Holder$, a type subsuming a variety of unordered collection types. The specification needs to be strong enough to allow useful procedures to be written on it. At the same time, the specification needs to be general enough to allow both Set and $MultiSet$ to be subtypes of $Holder$. Our solution has the property that the specifications of $Holder$ methods are unchanged in Set and $MultiSet$. Others have proposed that subtypes must be able to strengthen the specifications of supertype methods; our solution suggests that perhaps this isn't necessary.

0 Holders

The problem is inspired by [5]. We assume a type $Data$, and deal with "holders", sets, and multisets of $Data$ objects. We want type $Holder$ to have methods for

- adding an element to the holder,
- removing an element from the holder,
- checking whether the holder contains a given element, and
- retrieving the size of, *i.e.*, number of elements in, the holder.

These methods are to be specified using the following trait.

$HolderTrait$: **trait**

introduces

$new: \rightarrow Holder$
$ins: Holder \times Data \rightarrow Holder$
$del: Holder \times Data \rightarrow Holder$
$mem: Holder \times Data \rightarrow \mathbb{B}$
$measure: Holder \rightarrow \mathbb{N}$

asserts
  generated by new, ins
  $\langle\, \forall\, h\colon Holder,\ d, d0, d1\colon Data \,\triangleright$
    $ins(ins(h, d0), d1) = ins(ins(h, d1), d0)$ % ins is symmetric
    $\neg mem(new, d)$
    $mem(ins(h, d0), d1) \equiv d0 = d1 \vee mem(h, d1)$
    $mem(h, d) \Rightarrow ins(del(h, d), d) = h$
    $\neg mem(h, d) \Rightarrow del(h, d) = h$
    $mem(h, d) \Rightarrow measure(del(h, d)) < measure(h)$
  $\rangle$

The **generated by** clause is equivalent to asserting

$$\langle\, \forall\, h\colon Holder \,\triangleright\, h = new() \ \vee\ \langle\, \exists\, h'\colon Holder,\ d\colon Data \,\triangleright\, h = ins(h', d)\ \rangle\rangle \quad . \qquad (0)$$

Subtypes are to inherit the method specifications of $Holder$. However, clients should be able to infer stronger properties about the functions introduced in trait $HolderTrait$ than what is being asserted in that trait. For example, the Set subtype should be able to add the axioms

$ins(h, d) = ins(ins(h, d), d)$ % ins is idempotent
$mem(del(h, d0), d1) \equiv d0 \neq d1 \wedge mem(h, d1)$
$measure(h) = \langle\, \#\, d \mid mem(h, d) \,\triangleright\, d\ \rangle$.

The expression $\langle\, \#\, d \mid mem(h, d) \,\triangleright\, d\ \rangle$ denotes the number of d's for which $mem(h, d)$ holds. These axioms do not hold for multisets, so $MultiSet$ must be able to extend the axiom set of $HolderTrait$ in a different way.

1 A first attempt

In this section, we discuss a failed attempt at a solution. Let Set be a subtype of $Holder$. We consider *extending* the trait $HolderTrait$ to a trait $SetTrait$, which will be the trait used by type Set. By extending a trait, we mean asserting additional properties.

$SetTrait$: **trait**
extends $HolderTrait$

asserts
  $\langle\, \forall\, s\colon Set,\ d, d0, d1\colon Data \,\triangleright$
    $ins(s, d) = ins(ins(s, d), d)$
    $mem(del(s, d0), d1) \equiv d0 \neq d1 \wedge mem(s, d1)$
    $measure(s) = \langle\, \#\, d \mid mem(s, d) \,\triangleright\, d\ \rangle$
  $\rangle$

Note that the quantification in the **asserts** section of trait $SetTrait$ states properties pertaining only to sets, not to holders in general. The intention is that by asserting these properties only for sets, multisets are unaffected.

We proceed to demonstrate the above guess falls short. We do so by proving

Either there is no set whose $measure$ is 0, or ins is idempotent for every holder. (1)

In either case, this is not what we're after. If no set measures 0, there is no representation of the empty set; if ins is idempotent, $MultiSet$ cannot be a subtype of $Holder$. The rest of this section is devoted to the proof of this property.

First, let's show

$$\langle\, \forall\, s{:}\, Set \;\triangleright\; measure(s) = 0 \;\Rightarrow\; new() = s \,\rangle \qquad . \tag{2}$$

Let s be a Set element. We need to show

$$measure(s) = 0 \;\Rightarrow\; new() = s \qquad . \tag{3}$$

Since Set is a subtype of $Holder$, s is a $Holder$ element, a fact we will denote $s : Holder$. We instantiate (0) with $h := s$ and get

$$s = new() \;\;\vee\;\; \langle\, \exists\, h{:}\, Holder,\; d{:}\, Data \;\triangleright\; s = ins(h, d) \,\rangle \qquad .$$

The first disjunct immediately shows (3); we show that the second disjunct also leads to (3). For s of the form $ins(h, data)$, where h is some holder and $data$ some data element, we calculate,

$$\begin{array}{ll}
 & measure(s) = 0 \\
= & \qquad \{\;\; \text{def. of } measure \text{, using } SetTrait \text{ since } s : Set \;\;\} \\
 & \langle\, \#\, d \mid mem(s, d) \;\triangleright\; d \,\rangle = 0 \\
= & \qquad \{\;\; \# \text{ and } \forall \;\;\} \\
 & \langle\, \forall\, d \;\triangleright\; \neg mem(s, d) \,\rangle \\
\Rightarrow & \qquad \{\;\; \text{instantiate, with } d := data \;\;\} \\
 & \neg mem(s, data) \\
= & \qquad \{\;\; s = ins(h, data) \;\;\} \\
 & \neg mem(ins(h, data), data) \\
= & \qquad \{\;\; \text{def. of } mem \text{, using } HolderTrait \;\;\} \\
 & \neg(data = data \;\vee\; mem(h, data)) \\
= & \qquad \{\;\; \text{predicate calculus} \;\;\} \\
 & false \\
\Rightarrow & \qquad \{\;\; \text{predicate calculus} \;\;\} \\
 & s = new() \qquad .
\end{array}$$

We have thus proven (2).

Because new is a function, we conclude from (2) that

$$\langle\, \exists\, s\colon Set \;\triangleright\; measure(s) = 0 \,\rangle \;\Rightarrow\; new() : Set \qquad . \tag{4}$$

We now show that if there is a set that measures 0, then ins is idempotent for every holder, *i.e.*,

$$\begin{array}{l} \langle\, \exists\, s\colon Set \;\triangleright\; measure(s) = 0 \,\rangle \;\Rightarrow \\ \quad \langle\, \forall\, h\colon Holder,\; d\colon Data \;\triangleright\; ins(h,d) = ins(ins(h,d),d) \,\rangle \qquad . \end{array} \tag{5}$$

Note that this is exactly (1), which we have set out to prove. Because of (4), (5) follows from

$$new() : Set \;\Rightarrow\; \langle\, \forall\, h\colon Holder,\; d\colon Data \;\triangleright\; ins(h,d) = ins(ins(h,d),d) \,\rangle \qquad . \tag{6}$$

We assume the antecedent, and prove for every holder h and data element d,

$$ins(h,d) = ins(ins(h,d),d) \qquad . \tag{7}$$

Our proof is by induction on the number of applications of ins to generate h; induction is enabled by (0). If holder h is generated without any applications of ins, $h = new()$. From the antededent of (6), we then have $h : Set$. Thus, by the first axiom in $SetTrait$, (7) follows.

If holder h is generated by a positive number of applications of ins, there is some holder h' and data element d' such that

$$h = ins(h', d') \qquad .$$

We calculate,

$$\begin{array}{ll} & ins(h,d) \\ = & \quad \{\;\; h = ins(h',d') \;\;\} \\ & ins(ins(h',d'),d) \\ = & \quad \{\;\; \text{symmetry of } ins \text{, } \textit{i.e.}\text{, first axiom in } HolderTrait \;\;\} \\ & ins(ins(h',d),d') \\ = & \quad \{\;\; \text{induction hypothesis with } h := h' \text{, since } h' \text{ is constructed using fewer} \\ & \qquad \text{applications of } ins \text{ than } h \;\;\} \\ & ins(ins(ins(h',d),d),d') \\ = & \quad \{\;\; \text{symmetry of } ins \text{, twice} \;\;\} \\ & ins(ins(ins(h',d'),d),d) \\ = & \quad \{\;\; h = ins(h',d') \;\;\} \\ & ins(ins(h,d),d) \qquad . \end{array}$$

So, (7) holds also in the inductive step. We have thus showed (6), which, as previously remarked, equals (1). We conclude that the solution attempted in this section leads to something undesirable.

2 The solution

The problem with the attempt from the previous section is that the additional axioms that should apply only to sets make their way into the lives of all holders. We can thus no longer pretend to have just one axiom set which can be extended in subtypes. Instead, we decide to declare one axiom set per object.

2.0 Holder type declaration

We continue in the style of [3], and start by declaring type $Holder$.

$\textbf{type}\ Holder$

An attribute of a holder is its validity. This attribute is modeled in the usual way by introducing an abstract $Holder$ field $valid$.

$\textbf{spec var}\ valid\colon Holder \to \mathbb{B}$

Another attribute of a $Holder$ object is the holder (or set, multiset, or whatever) value it represents. For this, we use an abstract field $value$. We assume that $Value$ denotes the type "set of $Data$", and then declare $value$ as follow.

$\textbf{spec var}\ value\colon Holder \to Value$

Rather than using a trait declaration, we introduce functions new, ins, *etc.*, as abstract $Holder$ fields of the appropriate types.

$\textbf{spec var}\ new\colon Holder \to Value$
$\textbf{spec var}\ ins\colon Holder \to Value \times Data \to Value$
$\textbf{spec var}\ del\colon Holder \to Value \times Data \to Value$
$\textbf{spec var}\ mem\colon Holder \to Value \times Data \to \mathbb{B}$
$\textbf{spec var}\ measure\colon Holder \to Value \to \mathbb{N}$

Validity of a holder consists of two parts: the subtype-independent part and the subtype-specific part. We model the latter with a field $svalid$.

$\textbf{spec var}\ svalid\colon Holder \to \mathbb{B}$

We can now define the subtype-independent part by giving the representation of $valid$.

$$
\begin{array}{l}
\textbf{rep}\ valid[h\colon Holder] \equiv \\
\quad svalid[h] \wedge \\
\quad \langle \forall\, v\colon Value,\ d, d0, d1\colon Data \triangleright \\
\qquad ins[h](ins[h](v, d0), d1) = ins[h](ins[h](v, d1), d0) \\
\qquad \neg mem[h](new[h], d) \\
\qquad mem[h](ins[h](v, d0), d1) \equiv d0 = d1 \vee mem[h](v, d1) \\
\qquad mem[h](v, d) \Rightarrow ins[h](del[h](v, d), d) = v \\
\qquad \neg mem[h](v, d) \Rightarrow del[h](v, d) = v \\
\qquad mem[h](v, d) \Rightarrow measure[h](del[h](v, d)) < measure[h](v) \\
\quad \rangle
\end{array}
$$

This representation declaration requires the following dependency declarations (see [3, 4]).

$$\textbf{depends}\ valid[h\colon Holder]\ \textbf{on}\ svalid[h], new[h], ins[h], del[h], mem[h], measure[h]$$

For completeness, let us declare and specify the methods here. In these methods declarations, the first parameter is the "self" parameter.

$$
\begin{array}{l}
\textbf{method}\ add(h\colon Holder\ ;\ d\colon Data) \\
\quad \textbf{requires}\ valid[h] \\
\quad \textbf{modifies}\ value[h] \\
\quad \textbf{ensures}\ value_{post}[h] = ins[h](value_{pre}[h], d) \\
\textbf{method}\ remove(h\colon Holder\ ;\ d\colon Data) \\
\quad \textbf{requires}\ valid[h] \\
\quad \textbf{modifies}\ value[h] \\
\quad \textbf{ensures}\ value_{post}[h] = del[h](value_{pre}[h], d) \\
\textbf{method}\ member(h\colon Holder\ ;\ d\colon Data)\ \textbf{returns}\ (b\colon \mathbb{B}) \\
\quad \textbf{requires}\ valid[h] \\
\quad \textbf{ensures}\ b \equiv mem[h](value[h], d) \\
\textbf{method}\ size(h\colon Holder)\ \textbf{returns}\ (n\colon \mathbb{N}) \\
\quad \textbf{requires}\ valid[h] \\
\quad \textbf{ensures}\ n = measure[h](value[h])
\end{array}
$$

2.1 Subtype type declarations

We declare Set to be a subtype of $Holder$.

$$\textbf{type}\ Set <: Holder$$

Subtype Set reveals in a set-specific way the details of the functions declared as fields in type $Holder$.

$$\begin{array}{l}
\textbf{rep}\ new[s\colon Set] = \emptyset \\
\textbf{rep}\ ins[s\colon Set] = \langle\, \lambda\, v\colon Value,\ d\colon Data \triangleright\ v \cup \{d\}\ \rangle \\
\textbf{rep}\ del[s\colon Set] = \langle\, \lambda\, v\colon Value,\ d\colon Data \triangleright\ v \setminus \{d\}\ \rangle \\
\textbf{rep}\ mem[s\colon Set] = \langle\, \lambda\, v\colon Value,\ d\colon Data \triangleright\ d \in v\ \rangle \\
\textbf{rep}\ measure[s\colon Set] = \langle\, \lambda\, v\colon Value \triangleright\ |v|\ \rangle
\end{array}$$

Here, $\emptyset$, $\cup$, $\setminus$, $\in$, and $|\,|$ are taken from some trait for $Value$, *i.e.*, sets of $Data$ elements. Since these representations are consistent with the axioms given in the representation of $valid$, we have not blown our chances to produce valid Set objects. For an example of how validity is established, see the example in Section 3.2 below.

3 Examples

In this section, we give four examples. The first shows an example $Holder$ client. The second demonstrates how axioms can be introduced in more than one place. The third takes an example down to the implementation, and shows how validity of an object can be established. The last example is taken from [5] and shows how our approach differs from that in [5].

3.0 A holder client

We give an example of a $Holder$ client. Consider a procedure $AddThree$ that adds three given elements to a holder. Its specification is given as

$$\begin{array}{l}
\textbf{proc}\ AddThree(h\colon Holder\ ;\ d0, d1, d2\colon Data) \\
\quad \textbf{requires}\ valid[h] \\
\quad \textbf{modifies}\ value[h] \\
\quad \textbf{ensures}\ value_{post}[h] = ins[h](ins[h](ins[h](value_{pre}[h], d0), d1), d2) \qquad .
\end{array}$$

Its implementation is simply

$$\textbf{call}\ add(h, d0)\ ;\ \textbf{call}\ add(h, d1)\ ;\ \textbf{call}\ add(h, d2) \qquad ,$$

whose proof is straightforward. Because it is known that $ins[h]$ is symmetric for any holder h, the implementation could also have been given, for example, as

$$\textbf{call}\ add(h, d2)\ ;\ \textbf{call}\ add(h, d1)\ ;\ \textbf{call}\ add(h, d0) \qquad .$$

Clients of $AddThree$ may know that the holder they pass in is of a particular subtype of $Holder$. Clients should be able to benefit from that additional information. For example, one should be able to prove the Hoare triple

$$\begin{array}{ll} \{ & s : Set \wedge valid[s] \wedge measure[s](value[s]) = 0 \quad \} \\ & \quad \textbf{call } AddThree(s, d, d, d) \\ \{ & \quad measure[s](value[s]) = 1 \qquad\qquad\quad \} \end{array} \quad .$$

We end this example by showing the proof of this in full detail.

$$\begin{array}{ll} & wlp.(\textbf{call } AddThree(s, d, d, d)).(measure[s](value[s]) = 1) \\ = & \quad \{ \quad wlp \text{ of } \textbf{call} \quad \} \\ & valid[s] \wedge \\ & \langle \forall\, value' \mid value'[s] = ins[s](ins[s](ins[s](value[s], d), d), d) \wedge \\ & \qquad \langle \forall\, h{:}\, Holder \mid h \neq s \triangleright value'[h] = value[h] \rangle \triangleright \\ & \quad measure[s](value'[s]) = 1 \rangle \\ \Leftarrow & \quad \{ \quad \text{predicate calculus} \quad \} \\ & s : S \wedge valid[s] \wedge \\ & \langle \forall\, value' \mid value'[s] = ins[s](ins[s](ins[s](value[s], d), d), d) \triangleright \\ & \quad measure[s](value'[s]) = 1 \rangle \\ = & \quad \{ \quad \text{equals for equals} \quad \} \\ & s : S \wedge valid[s] \wedge \\ & \langle \forall\, value' \mid value'[s] = ins[s](ins[s](ins[s](value[s], d), d), d) \triangleright \\ & \quad measure[s](ins[s](ins[s](ins[s](value[s], d), d), d)) = 1 \rangle \\ = & \quad \{ \quad \text{range of } value' \text{ is nonempty} \quad \} \\ & s : S \wedge valid[s] \wedge measure[s](ins[s](ins[s](ins[s](value[s], d), d), d)) = 1 \\ = & \quad \{ \quad s : S \wedge valid[s] \text{, and } Set \ \textbf{rep} \text{ clauses for } measure \text{ and } ins \quad \} \\ & s : S \wedge valid[s] \wedge |value[s] \cup \{d\} \cup \{d\} \cup \{d\}| = 1 \\ \Leftarrow & \quad \{ \quad \text{properties of } \cup, |\,|, \text{ and } \emptyset \quad \} \\ & s : S \wedge valid[s] \wedge |value[s]| = 0 \\ = & \quad \{ \quad s : S \wedge valid[s] \text{, and } Set \ \textbf{rep} \text{ clause for } measure \quad \} \\ & s : S \wedge valid[s] \wedge measure[s](value[s]) = 0 \end{array}$$

3.1 Different axioms

In the $Holder$ and Set example, some general properties of the new, ins, *etc.*, functions are provided for all $Holder$ elements. In the Set subtype, we chose to give the representations of the functions directly rather than providing more axioms. We now give an example where

the representation of a function is postponed further, but where subsequent subtypes provide additional axioms.

Consider a type $BinRel$ of binary relations over $Data$ elements. Some binary relations are pre-orders, some are partial orders, some are total orders, *etc.* Moreover, every partial order is a pre-order, for example, so we can consider letting a type of partial orders be a subtype of the type of pre-orders. Let's define the supertype of these types, $BinRel$.

type $BinRel$
spec var $valid\colon BinRel \to \mathbb{B}$
spec var $op\colon BinRel \to Data \times Data \to \mathbb{B}$

Sample methods of $BinRel$ are

method $eval(r\colon BinRel\ ;\ x, y\colon Data)$ **returns** $(b\colon \mathbb{B})$
  requires $valid[r]$
  ensures $b \equiv op[r](x, y)$,

which evaluates op for some given arguments, and

method $lhs(r\colon BinRel\ ;\ y\colon Data)$ **returns** $(s\colon$ **set of** $Data)$
  requires $valid[r]$
  ensures $s = \{\ x \mid op[r](x, y) \triangleright x\ \}$,

which returns the set of x's that are left-related via op to y.

Validity of a binary relation is defined by the subtypes of $BinRel$. So that validity can encompass properties of op, we let $valid$ depend on op.

depends $valid[r\colon BinRel]$ **on** op

A subtype of $BinRel$ is $PreOrder$, which asserts that its op is reflexive and transitive. So that subtypes of $PreOrder$ can further constrain validity, we introduce another abstract field, $svalid$, on which $valid$ is declared to depend.

spec var $svalid\colon PreOrder \to \mathbb{B}$
depends $valid[pre\colon PreOrder]$ **on** $svalid[pre]$
rep $valid[pre\colon PreOrder] \equiv$
  $svalid[pre] \wedge$
  $\langle\ \forall\, x\colon Data \triangleright op[r](x, x)\ \rangle \wedge$ % reflexivity
  $\langle\ \forall\, x, y, z\colon Data \triangleright op[r](x, y) \wedge op[r](y, z) \Rightarrow op[r](x, z)\ \rangle$ % transitivity

Allowing subtypes of $PreOrder$ to further restrict op in their **rep** for $svalid$ mandates that one declare a dependency of $svalid$ on op. But to satisfy the authenticity requirement

(see [3]), such a dependency needs to be placed near the declaration of op. Since we don't assume that $BinRel$ and $PreOrder$ are declared in the same modular unit, we will employ an (ugly) trick.

> **Remark.** Other solutions exist, however. An attractive one is to allow non-functional **rep** clauses and allow each abstract field to have more than one (non-functional) **rep** clause per object (see [0]). Another possibility is to realize that the representation of op is constant for any particular object. Constant is a special case of being monotonic, so the ideas for stretching the authenticity requirement presented in [1] might well be applicable. ∎

The trick is to introduce a new name for op, call it sop.

> **spec var** $sop: PreOrder \to Data \times Data \to \mathbb{B}$
> **depends** $op[pre: PreOrder]$ **on** $sop[pre]$
> **rep** $op[pre: PreOrder] = sop[pre]$

We can now declare a dependency of $svalid$ on sop.

> **depends** $svalid[pre: PreOrder]$ **on** $sop[pre]$

Let's declare one more subtype. A partial order is pre-order that is also antisymmetric.

> **type** $PartialOrder <: PreOrder$
>
> **spec var** $tvalid: PartialOrder \to \mathbb{B}$
> **depends** $svalid[po: PartialOrder]$ **on** $tvalid[po]$
> **rep** $svalid[po: PartialOrder] \equiv$
> $tvalid[po] \wedge$
> $\langle \forall\, x, y: Data \triangleright\ sop[r](x, y) \wedge sop[r](y, x) \Rightarrow x = y \rangle$ % antisymmetry
> % introduce yet another name for op
> **spec var** $top: PartialOrder \to Data \times Data \to \mathbb{B}$
> **depends** $sop[po: PartialOrder]$ **on** $top[po]$
> **rep** $sop[po: PartialOrder] = top[po]$

To recap, we have shown an example of subtypes that give additional axioms about a function op without revealing the exact value of op. An exact value of op is provided in our next example.

3.2 Establishing validity

We build on the $BinRel$ type defined in Section 3.1 and give an implementation of one of its subtypes.

To give an example of a particular partial order, let's assume that $Data$ is the type

record a, b: $I\!N$ **end**

We declare a type $PointwiseOrder$.

type $PointwiseOrder <: PartialOrder$
rep $top[ptw: PointwiseOrder] = \langle\, \lambda\, x, y: Data \triangleright\ x.a \leq y.a \wedge x.b \leq y.b \,\rangle$

We give a particular implementation of $PointwiseOrder$, call it $PtwOrdImpl$. This implementation caches a closure.

type $PtwOrdImpl <: PointwiseOrder$
var $d: PtwOrdImpl \rightarrow Data$
var $cl: PtwOrdImpl \rightarrow$ **set of** $Data$
depends $tvalid[ptw: PtwOrdImpl]$ **on** $d[ptw], cl[ptw]$
rep $tvalid[ptw: PtwOrdImpl] \equiv cl[ptw] = \{\, x \mid op[ptw](x, d[ptw]) \triangleright\ x \,\}$

So that clients can initialize $PtwOrdImpl$ objects to be valid, there is an $init$ method.

method $init(ptw: PtwOrdImpl)$
  modifies $valid[ptw]$
  ensures $valid_{post}[ptw]$

Lastly, we show the implementations of $PtwOrdImpl$'s methods.

method impl $init(ptw: PtwOrdImpl)$
  $d[ptw] := Data(a := 0, b := 0)$;
  $cl[ptw] := \{d[ptw]\}$

To prove that $init$ establishes $valid[ptw]$, one needs to prove the right-hand side of the **rep** clauses of $valid$ and $svalid$ (shown in Section 3.1) and $tvalid$. That $tvalid[ptw]$ holds upon exit follows from the body of $init$. The other conjuncts of validity follow from the **rep** clauses of $valid$, $svalid$, op, sop, and top, in the way these apply to $PtwOrdImpl$ objects. Hence, if some inconsistency has been introduced in those **rep** clauses, $valid[ptw]$ can never be established.

We close by giving the implementations of $eval$ and lhs.

method impl $eval(ptw: PtwOrdImpl; x, y: Data)$ **returns** $(b: I\!B)$
  $b := x.a \leq y.a \wedge x.b \leq y.b$

```
method lhs(ptw: PtwOrdImpl ; y: Data) returns (s: set of Data)
  if d[ptw] = y then
    s := cl[ptw]
  else
    s := {} ;
    for i := 0 to y.a do
      for j := 0 to y.b do
        s := s ∪ {Data(a := i, b := j)}
      end
    end
  fi
```

3.3 Existentials

The introduction of the construct **existential** in [5] was motivated by an example *Fixpoint* class. We retell a slightly simplified version of that example here. Consider a trait

```
FnTrait : trait
introduces
  fn: ℕ → ℕ
```

that introduces a function symbol *fn* used in the following class:

```
class Fixpoint

method group
  existential fn
  method do_fn(x: ℕ) returns (r: ℕ)
    ensures r = fn(x)

method group
  method is_fixpoint(x: ℕ) returns (b: 𝔹)
    ensures b ≡ x = fn(x)

end Fixpoint      .
```

The implementation of method *is_fixpoint* invokes method *do_fn* in the obvious way. Now, [5] argues that clients should see only the above specification of, *e.g.*, method *do_fn*. This is to support modularity, because the only thing a client can assume about function *fn* is that it is a function over the natural numbers. The idea is then to let subclasses make use of different values for *fn*. According to [5], different subclasses of *Fixpoint* can assume different functions—they need show only that there exists a function for which the implementation meets the specification.

Consider, then, the following code snippet (also from [5]):

if $o1.do\_fn(0) = o2.do\_fn(0)$
  then $x := true$
  else $x := false$
fi .

Since to clients both invocations of do_fn are interpreted using the same function fn, clients are able to conclude that $x = true$ upon termination of this statement. But $o1$ and $o2$ may be of different subclasses of $Fixpoint$ and may thus have made use of different values for fn when proving their implementations, so executing the above statement may possibly result in $x = false$. Hence, modularity is actually not facilitated.

Let's take a look at how our approach handles this example. We would declare $Fixpoint$ as follows.

type $Fixpoint$

spec var $fn: Fixpoint \to \mathbb{N} \to \mathbb{N}$

method $do\_fn(fp: Fixpoint\ ;\ x: \mathbb{N})$ **returns** $(r: \mathbb{N})$
  $r = fn[fp](x)$

method $is\_fixpoint(fp: Fixpoint\ ;\ x: \mathbb{N})$ **returns** $(b: \mathbb{B})$
  $b \equiv x = fn[fp](x)$

The implementation of method $is\_fixpoint$ can invoke do_fn and can then be proven.

In the code snippet above, the final value of x will be

$$fn[o1](0) = fn[o2](0) \quad .$$

Knowing about $o1$ and $o2$ only that they are $Fixpoint$ objects, one cannot simplify this expression further, because the functions $fn[o1]$ and $fn[o2]$ may differ. Thus, our approach leads to the desired result.

4 Conclusions

We achieved our goal of writing a specification for type $Holder$, permitting both Set and $MultiSet$ as subtypes. (In fact, our approach even permits a $Holder$ subtype that sometimes keeps track of duplicate elements and sometimes does not, according to some switch that clients can modify at run-time.) The $Holder$ specification itself was strong enough to allow different implementations of the example procedure $AddThree$. The specifications of $Holder$'s methods did not need to be changed in subtypes.

In our course of action, we successfully combined Larch-like algebraic specifications [2] and the approach in [3]. The intent was not to debate a two-tier approach —it just happened that the particular notation we chose in this note did not reveal specifications as such— but rather to show that specifications written in the style of [3] can be given algebraically.

The most important idea we presented is the way in which subtypes can extend an axiom set. The means for doing so comes directly from [3], and hence the approach has a solid formal foundation.

References

[0] D.L. Detlefs and K.R.M. Leino. Computing dependencies. KRML 61, Digital's Systems Research Center, November 1995.

[1] D.L. Detlefs and K.R.M. Leino. Specifying the state of modules. KRML 57, Digital's Systems Research Center, September 1995.

[2] J.V. Guttag, J.J. Horning, and J.M. Wing. The Larch family of specification languages. *IEEE Software*, 2(5):24–36, September 1985.

[3] K.R.M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.

[4] K.R.M. Leino and G. Nelson. Beyond stacks. KRML 54, Digital's Systems Research Center, July 1995.

[5] R. Stata and J.V. Guttag. Modular reasoning in the presence of subclassing. *ACM SIGPLAN Notices*, 30(10):200–214, October 1995. OOPSLA '95 conference proceedings.

A A different solution to the holder problem

This note presented a algebraic specification of a type $Holder$. In this appendix, we record a different specification that does not make use of functions and axioms like the previously presented solution does.

type $Holder$

spec var $valid: Holder \to \mathbb{B}$
spec var $contents: Holder \to Data \to \mathbb{N}$
spec var $capacity: Holder \to \mathbb{N}$

method $add(h\colon Holder\ ;\ d\colon Data)$
  requires $valid[h]$
  modifies $contents[h][d]$
  ensures $contents_{post}[h][d] = min(contents_{pre}[h][d] + 1\,,\, capacity[h])$

method $remove(h\colon Holder\ ;\ d\colon Data)$
  requires $valid[h]$
  modifies $contents[h][d]$
  ensures $contents_{post}[h][d] = max(contents_{pre}[h][d] - 1\,,\, 0)$

method $member(h\colon Holder\ ;\ d\colon Data)$ **returns** $(b\colon I\!B)$
  requires $valid[h]$
  ensures $b \equiv contents[h][d] \neq 0$

method $size(h\colon Holder)$ **returns** $(n\colon I\!N)$
  requires $valid[h]$
  ensures $n = \langle\, \Sigma\ d\colon Data \triangleright\ contents[h][d]\ \rangle$

If the specification language at hand does not feature a Σ quantifier, one can instead introduce another field

spec var $cardinality\colon Holder \to I\!N$,

which the add and $remove$ methods update appropriately.

The Set and $MultiSet$ subtypes distinguish themselves in the following way.

type $Set <: Holder$
rep $capacity[s\colon Set] = 1$

type $MultiSet <: Holder$
rep $capacity[m\colon MultiSet] = \infty$

A final observation. The approach presented in this appendix lends itself to writing the following specification.

method $reduce(h\colon Holder\ ;\ d\colon Data)$
  requires $valid[h]$
  modifies $contents[h][d]$
  ensures $contents_{post}[h][d] \leq contents_{pre}[h][d]$

This specification cannot be written using the set of axioms from $HolderTrait$ in Section 0.