Modeling subtypes with only one object type

K. Rustan M. Leino
1 August 1995
Digital's Systems Research Center
130 Lytton Ave., Palo Alto, CA 94301, U.S.A.
rustan@pa.dec.com

The semantics of object types can be rather bewildering. In contrast, a language with only simple types like $\mathbf{int}$ and $\mathbf{bool}$ is quite manageable. In this note, we consider an object-oriented language with simple types, including a simple object type $\mathbf{obj}$. We then extend this language with sugar that models different types, subtyping, and also the $\mathbf{narrow}$ construct.

0 A simple language

Types

We consider a language where values can have types like $\mathbf{int}$ and $\mathbf{bool}$, called *simple* types. The language provides a special simple type called $\mathbf{obj}$. Values of this type are called *objects*. There is only one object constant, *viz.*, $\mathbf{nil}$; other objects are created by $\mathbf{new}$, described below.

Statements

Our language consists of usual imperative programming constructs like assignment, sequential composition, and block statements.

Fields

Fields of objects are declared as global *map* variables. A map variable has type $S \to T$ for some S and T. The type $S \to T$, being composed of other types, is not a simple type. For example, the declaration

$$\mathbf{var}\ f\colon \mathbf{obj} \to S \qquad ,$$

where S is a simple type, declares a field f. Every field is declared as a map from $\mathbf{obj}$.

A field can be dereferenced at an object o, written $f[o]$ (in many languages written $o.f$). One can use this dereference in two statements:

$$x := f[o] \qquad \text{and} \qquad f[o] := x \qquad .$$

The first of these sets x to the value of f at o, and the latter stores x in f at o. According to its type, a field can be dereferenced by any object. However, we make a restriction that **nil** may not be used. That is, a precondition of both statements above is $o \neq \textbf{nil}$.

Methods

A method m is declared (and given a specification) with a declaration like

$$\begin{array}{l} \textbf{method}\ r\colon R := t.m(a\colon A) \\ \quad \textbf{requires}\ \mathit{Pre} \\ \quad \textbf{modifies}\ \mathit{frame} \\ \quad \textbf{ensures}\ \mathit{Post} \qquad , \end{array}$$

where r of simple type R is a list of formal out-parameters of m, t (implicitly of type $\textbf{obj}$) denotes the formal "self" object (the object on which the method is invoked), a of simple type A is the list of formal in-parameters, predicate Pre is the precondition of the method, frame is a list of variables that are allowed to be modified by the method, and predicate Post specifies the postcondition of the method. In this note, we won't dwell more on the exact meaning of the specification; we assume the reader to have a feel for the meaning of "pre-" and "postconditions". We assume each method to have exactly one specification.

An invocation of a method m is written

$$v := o.m(x) \qquad ,$$

where v is a list of actual out-parameters, o is the actual "self" object, and x is a list of actual in-parameters. Like field dereferences, methods can be invoked only on non-**nil** objects. To capture that fact, we treat the aforementioned specification of m as sugar for

$$\textbf{requires}\ \mathit{Pre} \land t \neq \textbf{nil} \quad \textbf{modifies}\ \mathit{frame} \quad \textbf{ensures}\ \mathit{Post} \qquad .$$

Allocating new objects

New objects can be allocated using $\textbf{new}$. The programming system provides a special map variable alloc.

$$\textbf{var}\ \mathit{alloc}\colon \textbf{obj} \to \textbf{bool}$$

User programs cannot modify $alloc$ directly; only invocations of $\mathbf{new}$ alter the value of $alloc$.
The specification of $\mathbf{new}$ is in terms of $alloc$.

$$
\begin{aligned}
&x\colon \mathbf{obj} := \mathbf{new}() \\
&\quad \mathbf{requires}\ \mathit{true} \\
&\quad \mathbf{modifies}\ \mathit{alloc} \\
&\quad \mathbf{ensures}\ x \neq \mathbf{nil} \wedge \neg alloc_0[x] \wedge alloc[x] \wedge \langle \forall\, y \mid y \neq x \triangleright alloc_0[y] \Rightarrow alloc[y] \rangle
\end{aligned}
$$

Subscripting of a map with 0 indicates the initial value of that map.

1 Modeling multiple object types

User defined types

We now add to our language a notion of *user defined* object types. Such a type T is declared by

$$\mathbf{type}\ T <: S \qquad ,$$

where S is a (possibly user defined) object type. The *subtype* relation (also written $<:$) is the ordering defined as the reflexive, transitive closure of the $<:$ given in declarations of user defined types.

Type maps

For each object type T, the programming system provides a special map variable $\$T$.

$$\mathbf{var}\ \$T\colon \mathbf{obj} \rightarrow \mathbf{bool}$$

These variables are set appropriately (by the programming system) at the beginning of each execution of a program. They cannot subsequently be changed. The programming system guarantees the following about the values of type map variables.

$$\langle \forall\, T0, T1, o \mid T1 <: T0 \triangleright \$T1[o] \Rightarrow \$T0[o] \rangle$$

Allocating new objects

We now allow $\mathbf{new}$ to take the name of a type as an in-parameter. The specification of such invocations of $\mathbf{new}$ differ from the previous specification only in the last conjunct of the postcondition.

$$\begin{array}{l} x\colon \mathbf{obj} := \mathbf{new}(T) \\ \quad \mathbf{requires}\ \mathit{true} \\ \quad \mathbf{modifies}\ \mathit{alloc} \\ \quad \mathbf{ensures}\ x \neq \mathbf{nil} \wedge \neg \mathit{alloc}_0[x] \wedge \mathit{alloc}[x] \wedge \langle \forall\, y \mid y \neq x \triangleright \mathit{alloc}_0[y] \Rightarrow \mathit{alloc}[y] \rangle \\ \qquad\qquad\quad \wedge\ \$\,T[x] \end{array}$$

Fields

We now allow field declarations to be annotated with " $::\ T$ " for some object type T, as in

$$\mathbf{var}\ f\colon \mathbf{obj} \to S \qquad ::\ T \qquad ,$$

where, as before, S is a simple type. Recall, the evaluation of a field dereference $f[o]$ previously required $o \neq \mathbf{nil}$. For f annotated with type T, we now also require $\$\,T[o]$.

Methods

As for field declarations, we allow annotations of method declarations.

$$\begin{array}{l} \mathbf{method}\ r\colon R := t.m(a\colon A) \qquad ::\ T \\ \quad \mathbf{requires}\ \mathit{Pre} \\ \quad \mathbf{modifies}\ \mathit{frame} \\ \quad \mathbf{ensures}\ \mathit{Post} \end{array}$$

We take this specification to be sugar for

$$\mathbf{requires}\ \mathit{Pre} \wedge t \neq \mathbf{nil} \wedge \$\,T[t] \quad \mathbf{modifies}\ \mathit{frame} \quad \mathbf{ensures}\ \mathit{Post} \qquad .$$

Narrowing values

In the presence of map types, we can now define the operation $\mathbf{narrow}$. We do so by giving its specification, where T denotes the name of an object type.

$$\begin{array}{l} x\colon \mathbf{obj} := \mathbf{narrow}(y\colon \mathbf{obj},\ T) \\ \quad \mathbf{requires}\ \$\,T[y] \\ \quad \mathbf{modifies} \\ \quad \mathbf{ensures}\ x = y \end{array}$$

Note that $\mathbf{narrow}$ modifies nothing, and thus has no effect on the program state (other than on its out-parameter x, of course).

2 A comparison with Modula-3

We remark on a few differences between the object types presented here and those in Modula-3 [1].

Firstly, Modula-3 allows only single inheritance. We didn't need any such restriction for our object types.

Secondly, Modula-3 defines a notion of *assignable*. An object type T is assignable to an object type S exactly when $S <: T$ or $T <: S$. Let S and T denote object types for which neither $S <: T$ nor $T <: S$ is known. Then, for f a field of T and o an expression of type S, $f[o]$ fails to type check.

In what we have presented, the same restriction applies, but the error is not caught by the type checker, which insists only that o be of type $\mathbf{obj}$, but instead by the verifier, which will fail to prove that $\$T[o]$ holds.

Finally, Modula-3 gives a stronger specification of $x := \mathbf{new}(T)$. Where in our postcondition we wrote the conjunct

$$\$T[x] \qquad ,$$

Modula-3 says (in our notation)

$$\langle \forall\, T' \triangleright \$T'[x] \;\equiv\; T <: T' \,\rangle \qquad .$$

As an example, consider the declaration

$$\mathbf{type}\ \mathit{T1} <: \mathit{T0}$$

and the program snippet

$\mathbf{var}\ x \colon \mathbf{obj};\ \mathbf{begin}$
$\quad x := \mathbf{new}(\mathit{T0});$
$\quad \mathbf{if}\ \$\mathit{T1}[x]\ \mathbf{then}\ \mathit{wrong}\ \mathbf{else}\ \mathit{skip}\ \mathbf{fi}$
$\mathbf{end} \qquad .$

In Modula-3, this program is guaranteed to terminate, whereas one cannot prove the same for our programs.

3 Open arrays

We conclude by showing how open arrays are modeled.

Collections

Each open array type has an associated *collection* (terminology adapted from Euclid [0]). A collection is a map variable of the form

$$\mathbf{var}\ \mathit{array}\colon \mathbf{obj} \to \mathbf{nat} \to S \quad ,$$

where S is a simple type. The statements involving array are

$$x := \mathit{array}[o][i] \qquad \text{and} \qquad \mathit{array}[o][i] := x \quad .$$

(In Modula-3, $\mathit{array}[o][i]$ is written $o\uparrow[i]$ or simply $o[i]$, because the Modula-3 type system determines array uniquely from the type of o.) The first of these statements sets x to the value of element i of $\mathit{array}[o]$; the second sets element i of $\mathit{array}[o]$ to x. Index i must be a proper index into $\mathit{array}[o]$, as described below.

Allocating new open arrays

For each collection array, the programming system declares an associated map $\mathit{number\$array}$.

$$\mathbf{var}\ \mathit{number\$array}\colon \mathbf{obj} \to \mathbf{nat}$$

As for type maps, the maps associated with collections are set appropriately (by the programming system) at the beginning of each execution of a program, and are never changed thereafter.

We now introduce a new flavor of $\mathbf{new}$, tailored for use with open arrays. For array a collection, we specify the new $\mathbf{new}$ as

$$\begin{array}{l} x\colon \mathbf{obj} := \mathbf{new}(\mathit{array},\ n\colon \mathbf{nat}) \\ \quad \mathbf{requires}\ \mathit{true} \\ \quad \mathbf{modifies}\ \mathit{alloc} \\ \quad \mathbf{ensures}\ x \neq \mathbf{nil} \wedge \neg \mathit{alloc}_0[x] \wedge \mathit{alloc}[x] \wedge \langle \forall\, y \mid y \neq x \triangleright \mathit{alloc}_0[y] \Rightarrow \mathit{alloc}[y] \rangle \\ \qquad\qquad \wedge\ \mathit{number\$array}[x] = n \end{array}$$

Proper indices

Finally, we describe precisely what a "proper index" means. From the type system, we already have that the i in

$$array[o][i]$$

must be a natural number. Furthermore, in order for i to be a proper index, we require a precondition of

$$i < number\$array[o] \quad .$$

References

[0] B.W. Lampson, J.J. Horning, R.L. London, J.G. Mitchell, and G.J. Popek. Report on the programming language Euclid. Technical Report CSL-81-12, Xerox PARC, 3333 Coyote Hill Rd., Palo Alto, CA 94304, U.S.A., October 1981. An earlier version of this report appeared in ACM SIGPLAN Notices, 12(2), February 1977.

[1] G. Nelson, editor. *Systems Programming with Modula-3*. Series in Innovative Technology. Prentice-Hall, Englewood Cliffs, NJ, 1991.