Value Types in Eiffel

Stuart Kent and John Howse
Department of Computing, University of Brighton,
Brighton BN2 4GJ, UK.

Email: Stuart.Kent@brighton.ac.uk, John.Howse@brighton.ac.uk

Abstract

This paper proposes extensions to the semantics of Eiffel's expanded types to support the modelling of value types. This provides support for defining immutable entities and is useful in allowing ADT-style specifications of values to be given in Eiffel. An abstract semantics is given for expanded types in terms of order sorted first order predicate logic; this is less restrictive than the existing operational semantics, and allows value types to be identified directly with expanded types. An operational semantics, which is closer in spirit to the abstract semantics than the existing Eiffel semantics, is outlined. This retains most of the advantages of the abstract semantics. It is concluded that the extra flexibility thereby provided could be used to restructure the Eiffel data structure libraries by starting from ADT-style specifications of value types written directly in Eiffel.

1 Introduction

In object oriented (OO) programming and design it is possible to draw a distinction between value types and reference types. Entities (attributes, parameters, local program variables, etc.) of a reference type hold pointers to objects, which have state that may change over time, even though the entity itself does not change its value (which is a *pointer* to the object). On the other hand, entities of value type hold values directly, rather than pointers to values. In the words of Cook and Daniels [1994, p75] value types intrinsically are "immutable and lack identity".

This paper examines the modelling of value types in the OO programming language Eiffel. It identifies problems with the current model viz. expanded types, and proposes a small, backward compatible syntactic extension and a change to the semantics. The *ideal* or *abstract* semantics is described formally using first-order logic; an operational semantics, which is closer in spirit to this ideal than Meyer's semantics for expanded types, is then outlined.

Our interest in value types is motivated by the following observations.

1. Value types could be used more extensively, and effectively, in OO modelling and design. We believe that there are things which intrinsically are immutable and lack identity, and that these go beyond the basic types.[1] The example used in Section 2 illustrates this point.

2. It has proved difficult to provide complete specifications of the behaviour of objects and classes, in terms of Eiffel contracts, for certain data structures [McKim and Mondou, 1993]. Such specifications are important to ensure both correct data structure libraries and their correct use. We believe that the ability to define value types flexibly in the programming language provides

[1] Even if you disagree with this, you must agree that the claim can not be tested unless sufficient support is provided in programming and design languages for practioners to experiment with the technology.

a solution to this problem, a point which will be returned to in the section on further work.

Some support for modelling value types is already given in OO programming and design languages. Their role is recognised in the OO design method Syntropy [Cook and Daniels, 1994], and support provided to define them. Eiffel [Meyer, 1988] also includes some support for value types, more of which in Section 2. Programming languages such as C++ and Ada95 have value types, though this may be a leftover of their non-OO lineage. In Smalltalk, value types are approximated by immutable object types. However, with the exception of Syntropy this support is limited. In particular, it does not provide the flexibility afforded by algebraic specification languages [Guttag *et al.*, 1985], which, we believe, is necessary to support the aims of (1) and (2) above.

Value types are typically used to model basic types, such as the booleans and integers, but not much else. In the case of Eiffel, C++, Ada95 and Smalltalk, this may be because the support provided to build one's own value types has significant limitations. This paper will support this statement in detail for Eiffel. In the case of [Cook and Daniels, 1994], there is considerable support for defining value types. However, the authors observe (p379) that the use of value types available in design is often limited to those available in the target programming language.

We believe a solution to this problem is to enhance Eiffel. Our reasons are threefold.

1. Eiffel provides some support for formal specification in its assertion language, which as [Kent and Maung, 1995b] and this paper indicate, only requires relatively small extensions to give it the power of those specification languages mentioned, at least for sequential systems. This is important in respect of value types where most of the work in their definition has happened in the specification world.

2. Eiffel provides a seamless approach to development [Waldén and Nerson, 1994], where programming and specification are intermingled. This brings formal specification techniques within the realm of the practising software engineer, who is more likely to learn a programming language than a formal specification language. It also avoids the problem observed above of mapping a specification/design language (e.g. Syntropy) to a programming language.

3. Eiffel is a 'pure' object oriented language, in contrast to, for example, OO extensions of Z and VDM in the specification world, and C++ in the programming world. This also makes it a more suitable language than those mentioned, for the specification and implementation of an OO system.

The paper is organised as follows. Section 2 introduces expanded types in Eiffel by example, and section 3 notes their deficiencies (assuming the semantics of [Meyer, 1992]) when modelling value types. Section 4 gives an abstract semantics, providing an ideal model which it may or may not be possible to reproduce operationally. A formal approach is adopted to lend a degree of rigour to the discussion, and the semantics is shown to be compatible with, though not as constraining as, Meyer's semantics. This is used, in section 5, to argue that the semantics of expanded types in Eiffel may be relaxed in a way that allows the deficiencies noted in section 3 to be overcome, in particular, in the definition of generators, in allowing value subtyping, and in giving greater flexibility to the implementation of value types. Section 6 outlines how the abstract semantics may be reproduced operationally, making as few concessions as possible. Section 7 is a summary, section 8 discusses related work, and section 9 outlines further work.

2 Expanded types in Eiffel

An example of an expanded type, taken from a specification and implementation of draughts in Eiffel, is given by figure 1.

The purpose of *D_B_POS* is to provide a type whose values are valid positions on a draughts board. The value is given by the queries: *row* and *col* giving the coordinates of the position; *mid* identifying the position between two positions a square apart on a diagonal (useful when considering taking in draughts); *next* for giving the next position on the board (useful for step-

```
expanded class D_B_POS inherit
    BASIC_ROUTINES;
    D_B_POS_DIM

feature -- queries

    row, col: INTEGER;

    mid (to: D_B_POS): D_B_POS is
        require
            abs (row - to.row) = 2 and abs (col - to.col) = 2
        ensure
            Result.row = (row + to.row) // 2;
            Result.col = (col + to.col) // 2
        end;

    next: D_B_POS is
        require
            (col >= min_col and col <= max_col and row >= min_row and row <= max_row)
        ensure
            col = max_col implies (Result.col = min_col
                and (row = max_row implies Result.row = min_row)
                and (row /= max_row implies Result.row = row + 1));
            col /= max_col implies (Result.row = row and Result.col = col + 1))
        end;

feature -- command

    set (x, y: INTEGER) is
        require
            (x >= min_col and x <= max_col and y >= min_row and y <= max_row)
        ensure
            row = y;
            col = x
        end;

end -- class D_B_POS
```

Figure 1: Draughts board positions class

ping through each position e.g. when displaying the board). Note that *mid* is specific to draughts; it is required for identifying the position of a piece being taken. However, it would not be important for a game like reversi; instead other features would be required, e.g. for identifying whether a position is between two other positions in a particular direction. This illustrates the need to tailor a value type to a particular application.

set is a command for 'setting' a value to a given coordinate. To understand what is meant by 'set' here, it is necessary to briefly consider the semantics, which is described in full in [Meyer, 1992].

The difference, semantically, between expanded and reference types is centred around the meaning of equality and assignment. If a and b are entities of conforming reference types, then $a = b$ means that they contain the same object reference—i.e. they point to the same object. Similarly, $a := b$ means that the reference held by a is set to be the same as that held by b.

On the other hand, if a and b are of conforming expanded types (which in Eiffel means they must be of the same type), then $a = b$ means that the attributes of a must have the same contents as the attributes of b. Note that if some of these attributes hold references to objects then the semantics requires the references to be the same (i.e. the attributes in question to refer to the same object). An alternative would be a form of deep equality, where the references themselves do not need to be the same, but instead the values held in the attributes of the objects referred to must be the same. Similarly, assignment for expanded types directly copies the values stored in the attributes of b into the attributes of a, without going deeper into the object structure.

The semantics of reattachment (e.g. when parameters are passed) is the same as that for assignment in both cases. Eiffel also allows one of a and b to be of expanded type and the other of reference type. This paper will not consider these cases; see [Meyer, 1992] for details.

The semantics of expanded types effectively means that entities of value type hold the actual values (the record with the attributes as fields), rather than pointers to the values (as is the case for reference types). Thus, for example, the attribute

$$pos : D\_B\_POS$$

would cause the compiled code to store a record comprising a pair of integers, corresponding to the *row* and *col* attributes. The effect of performing the command $pos.set(x, y)$ would be to replace the value held by *pos* before the command with the value that results from performing the command, i.e. the pair of integers x and y. In contrast, if *D_B_POS* was a reference type, the effect of $pos.set(x, y)$ would not be to change the value held by *pos*, but rather the value of the object pointed to by *pos*.

3 Deficiencies of Expanded Types

D_B_POS is intended to be an example of a user-defined value type, as opposed to an Eiffel supplied value type, such as *INTEGER* or *BOOLEAN*. In this section we note some deficiencies in the use of expanded types to model user-defined value types, and look at possible work-arounds in the current language. In particular we observe in Section 3.1 that a backward-compatible change in syntax is required, and in Section 3.3 that a change to the semantics is required.

3.1 Constructors

We immediately notice that expanded types in Eiffel are missing some vital components that value types usually have. In particular they are missing a collection of *constructors*. In the world of algebraic specification these, together with transformers, are the operations which are used to generate the values of the type. For example, *one*, *two*, *three* etc. are constructors for the natural numbers; *succ* is a transformer; both are generators.

In order to provide a set of constructors for *D_B_POS* in Eiffel, it is necessary to go outside the class. For example, it is possible to define the class in figure 2, which is inherited into any class that needs the constructors.

```
class D_B_POS_CONSTR inherit
    D_B_POS_DIM

feature

    min_pos: D_B_POS is
        do
            Result.set (min_row, min_col)
        end;

    max_pos: D_B_POS is
        do
            Result.set (max_row, max_col)
        end;

    empty_pos: D_B_POS is
        do
            Result.set (0, 0)
        end;

    the_pos (x, y: INTEGER): D_B_POS is
        do
            Result.set (x, y)
        end;

end -- class D_B_POS_GEN
```

Figure 2: Constructors for *D_B_POS*

This solution is inelegant as it requires behaviour to be split across two classes, when really one should suffice. This in turn requires additional features (e.g. the *set* command) to be included in the class defining the value type, which are required in order to define the constructors.

It also means that the constructors may not be used in the definition of the value type itself, unless the class defining them is itself inherited into the class defining the value type. The consequence of not including the constructors is that other ways have to be found to specify certain aspects where constructors would be useful. Although this last point is not drastic it may mean that the same thing ends up being defined in two places, i.e. in the definition of the value type, and elsewhere in the definition of the constructors. It can also mean the inclusion of unnecessary extra baggage, such as *min_col*, *max_col* etc. in the value type definition.

The alternative is to inherit the constructors into the class defining the value type. However, this is inelegant as it means that values take on additional components (in the form of queries) which return other values that have absolutely nothing to do with the original value. For example, if *D_B_POS_CONS* were inherited into *D_B_POS* then it would be possible to perform the query ***pos.max_pos*** which returns a value that has nothing to do with the original value of *pos*.[2]

In section 5.1 we will show how constructors can be provided for expanded classes with a small but backward compatible extension to the syntax.

3.2 Subtyping

Another problem with expanded types in Eiffel is that they do not allow subtyping of value types. For example, suppose a class *PAIR_OF_INTEGER* has been defined with attributes *first* and *second* (renaming *row* and *col*) and command *set*, as in *D_B_POS*.

It seems reasonable that *D_B_POS* could be constructed by inheriting from this, and, furthermore, that any *D_B_POS* value could be used polymorphically as a *PAIR_OF_INTEGER* value. Although in Eiffel this inheritance is valid, the polymorphic substitution is not for expanded types.

3.3 Implementation of value types

We end this section by looking at the implementation of value types, when modelling them using Eiffel expanded types. This will illustrate a weakness in the current semantics for expanded types, namely that it is too restrictive.

Consider the definition of a stack value type which is outlined in figure 3.

The definition is based on the ADT for a stack which can be found in [Meyer, 1988, p55]. Re-

[2] In this respect, it is also interesting to note that the features inherited from *BASIC_ROUTINES* are not part of a pure definition of *D_B_POS*, and, indeed it would be better if the routines were provided as part of the definition of the type *INTEGER*, which is modelled as an expanded type in Eiffel.

```
expanded class STACK[X] creation
    new

feature -- queries
    pushed_stack(x: X): STACK[X]
        ensure
            Result.top = x
            Result.popped_stack = Current
            not Result.empty

    popped_stack: STACK[X]
        require
            not empty

    top: X
        require
            not empty

    empty: BOOLEAN

feature -- commands
    new
        ensure
            empty;

    push(x: X)
        ensure
            Current = pushed_stack

    pop
        require
            not empty
        ensure
            Current = popped_stack

end -- class STACK[X]
```

Figure 3: The value type stack defined as an expanded type

member this is a definition of a stack *value* type, as opposed to a reference type, and is the best translation of an ADT that we can manage in current Eiffel. In particular notice that the transformers *push* and *pop* are given both as queries (*popped_stack* and *pushed_stack*) and commands. The queries actually model the transformers; the commands are required to provide implementations of the queries to allow the latter to manipulate the stack values they return. Their appearance as queries means that they may appear in assertions, which is what is required to provide a translation of the ADT axioms appearing here in the ***ensure*** condition on *push_stack*.

The stack specification also illustrates the inelegance, in the definition of value types in Eiffel, of having to define extra commands to be able to implement the generators (in this case the transformers push and pop).

As it stands, the definition is only a specification because we have not said how the stack is to be implemented, in particular how the values of the stack are to be stored.[3]

Suppose we decide to implement the stack using a (resizeable) array, by inserting the following features, and defining the routines in terms of these.

```
feature{NONE}
    storage: ARRAY[X]
```

The bodies of *pushed_stack* and *push* would be defined, respectively, as follows:

```
    do
        !!Result
        Result.push(x)
    end

    do
        storage.resize(old storage.lower,
        old storage.upper + 1)
        storage.put(x,storage.upper)
    end
```

[3] In Eiffel, expanded classes can not be deferred, so it is necessary to provide the implementation. Even if this was not the case, then at some point we would have to provide an implementation.

The implementation of the *new* command, which is relevant to the ensuing discussion, is given below. The code resizes the array so that it contains no elements.

```
do
    storage.resize(old storage.lower,
    old storage.lower)
end
```

There would also be a creation routine to create the array.

The current semantics of expanded classes does not allow us to do this, at least not with the desired effects.

For example, suppose $s := t$, where s, t: $STACK[X]$, at some point in a program, then the value held by s becomes a reference to the array pointed to by t. Suppose that t is then updated by performing the *new* command. This has the effect of emptying the array by resizing it to zero. However, because s points to the same array, effectively it will also be updated, in that a query (say *top*) on s will give the same value as the same query on t, i.e. it will be an invalid call. This is undesirable, as effectively it makes the value type behave as if it were a reference type: one can manipulate a value t, only to discover that effects are also felt in another value s.

A similar argument could be constructed for the commands *push* and *pop*.

The situation just described would not occur if the semantics of assignment were changed so that the contents of the array were copied, rather than the reference to the array copied. Such a change to the semantics of assignement would, in turn, require a change to the semantics of equivalence to mean that the contents of the array were the same rather than them being the same array.

However, having assignment copy the whole structure would also be undesirable. If X in *STACK[X]* was substituted for a reference type, then one would probably want the references contained in the array to be copied, not the objects pointed to by those references, which would be the case with *deep_copy*. Similarly for equivalence.

It is in fact possible to effectively provide one's own version of equivalence and assignment in Eiffel, by defining new features in the way described. Indeed there are kernel library functions, such as *copy*, *clone* and their deep counterparts to help you do this.

We take a slightly different view that the semantics of standard assignment and equivalence for value types is too strong, in general. The abstract semantics given in the next section embodies this view, and in section 5 we demonstrate that it supports the modelling of value types required. A solution to the "not too deep" copy/equivalence problem is outlined in section 6.

4 Abstract Semantics

This section describes an abstract semantics for expanded types. The semantics is a level of abstraction removed from an operational semantics (a conceptual view of the target of compilation), such as that provided in [Meyer, 1992]. This enables an 'ideal' picture of the semantics to be painted, and it is this which is used in section 5 to show that expanded types could provide better modelling of value types, at least at a specification level. Section 6 outlines an operational semantics which is less pessimistic than the current semantics for Eiffel, with respect to this ideal.

A formal approach is used to lend rigour and precision to the argument. The approach is similar to that of Larch [Guttag *et al.*, 1985], where a specification language is given a semantics essentially in terms of theories of order sorted first order predicate logic (OSFOPL). This has the advantage of being widely understood, and it is hoped that such a semantics could provide the basis for proof assistant and simulation tools, such as those described in [Jones *et al.*, 1991, Costa *et al.*, 1990].

4.1 Framework

The basic idea is to convert an Eiffel class interface into a theory of OSFOPL. This theory must not only characterise the behaviour of an arbitrary object of the class (i.e. viewing the class as a template) but also the identity and creation of objects. Object identity may be charac-

terised simply in terms of a sort C_{id} corresponding to the collection of all potential instances (i.e. those that have existed, exist, and will exist) of classes conforming to C. Objects have state which may change through time, where the view on this state and ways of changing it are provided through its features. To model this, a sort Σ of possible states is provided. It is assumed that this includes all possible states of a system, only some of which will be abstract states of an object in the system (hence one sort suffices for all class theories and combinations thereof). The values that objects of class C may have in a state are determined by the sort C_s.

To model existence, a boolean function *exists* ranging over object identities, is included. This must also range over Σ, as the existence of an object may change from state to state. So that a different predicate is not required for each class, the sort ANY_{id} is also introduced. This characterises the collection of all potential objects, no matter what the class, and C_{id} is a subsort.

The values that objects, whatever the class, may have in a state are modelled by the sort ANY_s. The function

$$state(\Sigma, ANY_{id}) : ANY_s$$

is used to dereference object identities, where $state(s, x)$ returns the value held in the state of x at s. For each class, there is also a sort C_s which is a subsort of ANY_s. In addition, the axiom

$$\forall s : \Sigma; x : C_{id} \cdot state(s, x) : C_s$$

included in the theory for C, ensures that appropriate values are assigned to the state of objects of C. To summarise, the theory for a class C may be presented as

theory C
includes ANY
sorts
C_{id}, C_s
functions
... – feature signatures
axioms
$C_{id} < Any_{id}$
$C_s < Any_s$
$\forall s : \Sigma; x : C_{id} \cdot state(s, x) : C_s$
$Void : C_{id}$
... – axioms for features, invariants etc.

where the line **includes** ANY, means that all the symbols and axioms of ANY are included in C. Here ANY is given by

theory ANY
sorts
ANY_{id}, ANY_s, Σ
functions
$exists(\Sigma, ANY_{id}) : BOOLEAN$
$state(\Sigma, ANY_{id}) : ANY_s$
$Void : ANY_{id}$
...
axioms
$\forall s : \Sigma \cdot exists(s, Void)$
...

where it is assumed the theory of $BOOLEAN$ is already defined in the usual way (as we assume the logic is already defined). Of course, with value types in the language, it would be possible to derive the theories from suitable specifications. We come back to this in the section on further work. Note that, in the theory ANY, $Void$ is treated as an object (which happens to have no behaviour) and the axiom embodies the assumption that $Void$ always exists.

From now on we will refer to C_{id} as the identity or reference sort of C, and C_s as the value sort.

4.2 Features

For simplicity, in giving a semantics to features of a class, we assume a distinction between commands and queries (see e.g. [McKim and Mondou, 1993]), and do not allow queries with side-effects (which are visible to the client, so e.g. it is possible to create a new object as part of a query).

Queries

A query `q(a:A):B` of class C introduces the selector function on C_s

$$q(C_s, A) : B$$

in the theory of C.

The behaviour of features is obtained by deriving axioms from the code and/or specification (require and ensure conditions, and the class invariant). For the purposes of this paper it is not necessary to show in detail how this is done. The derivation of axioms from the specification is described in [Kent and Maung, 1995b].

However, it is instructive to look at the translation of query invocations, which forms part of the general interpretation of Eiffel expressions. For Eiffel expression A, its translation is given by $A\langle s, s', v, v'\rangle$, where $s, s' : \Sigma$ and $v, v' : ANY_s$ are provided by the context in which the translation of A appears. The rules giving the translation of queries as they appear in $A\langle s, s', v, v'\rangle$, are given below. [4] An explanation of the rules follows.

1. any call to a query $p(B)$ of the current object is replaced by $p(v, B\langle s, s', v, v'\rangle)$
2. any call to a query of another object through a $X.p(B)$, is replaced by
   (a) $p(state(s', X\langle s, s', v, v'\rangle), B\langle s, s', v, v'\rangle)$, if X is of reference type
   (b) $p(X\langle s, s', v, v'\rangle, B\langle s, s', v, v'\rangle)$, if X is of expanded type
3. any call to the old value of a query ***old*** $p(B)$ of C is replaced by $p(v, B\langle s, s, v, v\rangle)$
4. any call to the old value of a query of another object through ***old*** $X.p(B)$, is replaced by
   (a) $p(state(s, X\langle s, s', v, v'\rangle), B\langle s, s, v, v\rangle)$, if X is of reference type
   (b) $p(X\langle s, s', v, v'\rangle, B\langle s, s, v, v\rangle)$, if X is of expanded type

(1) and (2) deal with the case when the query does not appear in an old expression; cases (3) and (4), when it does. For the purposes of this discussion, we can ignore (3) and (4)—they are included for the sake of completeness.

s, s' represent system states. s is only used in the translation of ***old*** expressions so may be ignored (similarly for v). That is, in (1) and (2) s' represents the system state in which the query is being evaluated. (1) ensures that selector functions (which represent queries) of the same class are related on the same values (represented here by v').

(2.a), which deals with the case when the entity being invoked is of reference type, corresponds to invoking the selector function p on $state(s', X\langle s, s', v, v'\rangle)$, the value of the state of the object referenced by X at s', with argument $B\langle s, s', v, v'\rangle$. This effectively dereferences the pointer held by X, and then uses the appropriate selector function to invoke the query on the value returned.

(2.b) deals with the case when X is of expanded type, and is the same as (2.a) except that no dereferencing is required. This is because entities of expanded type are assumed to denote values directly (here represented by the value sorts), as opposed to pointers to those values.

Notice that the semantics makes no distinction between attributes and queries. The distinction is not appropriate at this level; rather it is a matter for the operational semantics. In what follows we will argue that the operational semantics currently given to Eiffel is too strong in its encoding of this ideal semantics (why this is ideal will be discussed in section 5). In section 6, we will outline an alternative approach to the operational semantics.

Commands

A command $c(a\colon A)$ introduces the function

$$c(\Sigma, C_s, A_{id}) : \Sigma \times C_s$$

As with queries, the detailed derivation of behavioural axioms is not important for this paper. The rules giving the translation of command invocations are given below, where: s, s' are the system states respectively before and after the invocation of the command; w, w' are the values of the current object in s, s', respectively; snd returns the second component of a pair.

1. any call to a command $c(A)$ of the current object is replaced by $v' = snd(c(v, A\langle s, s, v, v\rangle)$
2. any call to a command of another object through $X.c(A)$ translates to

[4] For simplicity, the rules only consider queries with a single argument, though they extend to queries with an arbitrary number of arguments. The rules including expressions of expanded type are given later.

(a) $state(s', X(s', s', v', v')) =$
$snd(c(s, state(s, X(s, s, w, w)),$
$A\langle s, s, w, w\rangle))$, if X is of reference type

(b) $X(s', s', w', w') =$
$snd(c(s, X(s, s, w, w), A\langle s, s, w, w\rangle)))$,
if x is of expanded type

(1) deals with the case of an object invoking one of its own commands. (2.a) deals with the case of an object's command being invoked through a pointer held in X, and dereferences as appropriate. (2.b) deals with the case where the object itself is held as a value denoted by the entity, in which case no dereferencing is necessary.

Thus the abstract semantics regards a command as returning a new state and a new value, where the latter is put in the appropriate place as part of the invocation of a command: as the value of an object pointed to by the entity involved in the invocation, for reference types, or as the value held by the entity itself for expanded types.

4.3 Equality and Assignment

Equality of expressions translates to equality of the translations of those expressions. In particular, for expressions of expanded types, this means comparing two values of the same value sort. Two entities of a (value) sort are equivalent iff the results returned by the selector functions, applied to each value for all arguments, are the same. This corresponds to query equivalence, making no distinction between derived queries and attributes.

Assignment translates to a statement ensuring that the denotation of the attribute in the l.h.s. after the assignment is equal to the denotation of the expression on the r.h.s. before the assignment, where for reference types, the denotation will be of reference sort, and for expanded types, of value sort, noting that in the latter case, equivalence of values is as above.

4.4 Compatibility with Expanded Semantics

The operational semantics for expanded types, described in [Meyer, 1992], is pessimistic with respect to the abstract semantics just described. The operational semantics of query and command invocation is compatible with the abstract semantics. According to the operational semantics of [Meyer, 1992], when a command is invoked on an entity of expanded type, its effect is to update the value, which is stored as a collection of attributes, denoted by that entity by performing the command directly on those attributes. This is compatible with our abstract view where the command returns a new value, possibly based on the old value, and the old value is then replaced by this.

Under the operational semantics $a = b$ for a and b of the same expanded type requires the attributes of a to be equivalent to the attributes of b. Given that all queries are calculated from the attributes, this guarantees that the queries are equivalent, which is all that is required by the semantics described above. However, as the example described in section 3.3 illustrates, situations are feasible where the (public) queries are equivalent, although the (secret) attributes are not. This indicates that the operational semantics is too strong i.e. pessimistic. (Although we have not done so above, we are at liberty to define a different notion of equality for each sort, and this would correspond to only requiring the public queries to be equivalent on all arguments.)

Similarly, for the assignment $a := b$ the current operational semantics requires the attributes of a to be the same as the attributes of b, which is stronger than the abstract semantics, which only requires the (public) queries on a to return the same values on all arguments as the queries on b.

5 Extension of expanded types to value types

5.1 Constructors

Abstractly, the semantics of commands means that for entity X of expanded type the invocation of a command $X.c(a)$ is achieved by constructing a new value, which is returned by the command, and then assigning this new value to X. This admits the possibility of allowing a command to appear on the right hand side of an assignment. In other words, $X.c(a)$ can be seen

as a shorthand for $X := X.c(a)$.

Applying this idea to *D_B_POS*, we observe that the invocation *pos.set(i,j)* could be a shorthand for *pos := pos.set(i,j)*. Applying the idea in reverse, observe that the assignment *pos := pos.next* would be required to make *pos* the next value in line from its current value. Given the semantics outlined in Section 4, as reflected by the existing operational semantics for expanded types, there is nothing to stop us defining *next* as the command

```
next is
    require
        (col >= min_col and col <= max_col
        and row >= min_row
        and row <= max_row)
    ensure
        col = max_col implies (col = min_col
            and (row = max_row implies
                row = min_row)
            and (row /= max_row implies
                row = old row + 1))
        col /= max_col implies (row = old row
            and col = old col + 1))
    end
```

and then write *pos.next* as a shorthand for the above assignment. In other words, commands of expanded types correspond to transformers from the ADT world.

So how does this help with the problem of defining constructors outlined in section 3. One further insight suffices. Looking at the definition of *set* in *D_B_POS*, one notices that it is not dependent on any old values. In which case *pos* is redundant in the r.h.s. of *pos := pos.set(i,j)*, and instead there is no reason why we should not be able to write *pos := set(i,j)* in this case. And here, *set* is behaving as an ADT constructor. Generalising this idea, we observe that the property of *set* we have identifed, is a property that any creation routine must have. Thus the constructors of a value type correspond directly to the creation commands of an expanded type.

To summarise, we propose that for entities of expanded type in Eiffel, commands may be used in expressions to return values of that type, and we have given an abstract semantics which supports this, and argued that, in this respect at least, the current operational semantics for Eiffel agrees with the abstract semantics. We further propose that creation commands may be used in expressions on their own (e.g. *set(i,j)* instead of *pos.set(i,j)*), thereby effecting constructors for the expanded type.

There are two further items to tidy up, in respect of the latter proposal:

1. Eiffel only allows one creation routine to be defined for expanded types, and that to have no arguments. We propose removing this restriction, by allowing a further keyword ***default*** to (optionally) label a single creation routine with no arguments which should be used, instead of the system default, to initialise attributes of value type when an object is created. For an example of these extensions in action see the rewrite of *STACK* in the section on further work. Note that this is backward compatible with existing syntax, in that if only one creation routine is included in the definition of the expanded type (currently all that is allowed), then this will be assumed to be the default and no extra annotation is required.

2. If *set* is also defined as a creation command in another value type, then it will not be clear to which value type a call to *set* refers. In such circumstances the name of the feature must be qualified by the name of the class as in, for example, *!D_B_POS!set(i,j)*.[5]

5.2 Subtyping

In the proposed semantics, subtyping of value types is embodied in an axiom which requires the value sort of the child to be a subsort of the value sort of the parent. For example, if *D_B_POS* is constructed by inheriting from *INTEGER_PAIR*, as suggested in section 3, then the axiom

$$D\_B\_POS_s < INTEGER\_PAIR_s$$

would appear in the theory for *D_B_POS*. This gives polymorphism as all features are factored

[5] We have used similar syntax to creation instructions in Eiffel, which use !...! to identify the exact type of object being created; here we wish to identify the exact type of value being created.

over the value sorts (in particular, queries correspond to selctor functions over the corresponding value sort).

This corresponds to the situation for reference types where, in our semantics, subtyping is achieved by having a similar axiom over the reference sorts.

The more sophisticated form of the semantics may need to be constructed to deal with renaming, in particular where repeated inheritance is concerned.

Unfortunately, as discussed in the next section, it is difficult to translate the above into an operational semantics, because of the way in which query equivalence must be modelled operationally. Fortunately, it seems that a useful compromise can be found.

5.3 Implementation

Since the semantics is based on query, rather than attribute equivalence, the problems identified in section 3 with regard to the implementation of value types disappear, at least at an abstract level. The trick is to find an operational semantics that gives the same flexibility as the abstract one. Such a semantics is outlined in the next section.

6 Operational Semantics

This section outlines an operational semantics which is less pessimistic than that for existing Eiffel with respect to the abstract semantics of section 4. This is only an outline and is yet to be tested. Some account is taken of the impact on efficiency.

6.1 Assignment

In section 3.3, we observed that a shallow copy of attributes (existing Eiffel semantics for assignment) might not allow the implementation of a value type using references to objects. The implementation of a stack value type using an array object was given as an example. It was argued that a form of "semi-deep" copy is required. The following scheme would, at first sight, seem to generalise on what was said there.

For all attributes A,

1. If A is of value type, copy the value.
2. If A is secret and of reference type, (shallow) copy the object referred to by A.
3. If A is public and of reference type, copy only the reference stored in A.

Note that public attributes are treated as in the existing semantics; this is needed to ensure query equivalence.

Returning to the stack example, the above scheme ensures that the array is copied, but not the objects referred to by the array. However, if the stack had been implemented using a linked list, this scheme would not have worked, as, in that case, we would also want to copy the individual links of the list, not references to them, otherwise we would end up with two lists interfering with one another. A possible solution to this problem is given by [Kent and Maung, 1995a] which introduces a notion of object ownership, and shows how it could be effected in Eiffel. The basic idea is to distinguish between objects wholly owned by another, and those which can be shared. In this case, owned objects would be copied, but not shared objects. In the case of a linked list, the elements of the list would be owned by the list so would be copied along with the list itself, but the objects referred to by elements of the list would be shared objects so only their references would be copied.

6.2 Query equivalence

The use of query equivalence, in favour of attribute equivalence, is in general not computable, as the queries on each value would have to be evaluated for all arguments, and, in general, this is not possible. However, a form of equivalence similar to the "semi-deep" copy described above would provide what is required, but without the drawbacks of simple attribute equivalence. That is two values of conformant type are considered equal if for all attributes A in common,

1. If A is of value type, the values stored in A for each object are equivalent.

2. If A is secret and of reference type, the object stored in A for one object is a (shallow) copy of that stored in A for the other object.

3. If A is public and of reference type, the references stored in A for each object are the same.

6.3 Subtyping

There is a slight complication with the proposed operational semantics, in respect of value subtyping, as the any form of deep equal, however shallow, requires the entities being compared to have the same attributes in the same places. With subtyping, it is possible for one value to have attributes that the other does not. It should be possible to obtain a limited form of subtyping and polymorphism on value types operationally, for example if it can be shown that one of the structures being compared is a substructure of the other. This would ensure that the values would be equivalent for the queries affected by that substructure. This corresponds to the case where the exact type of one of the operands is a subtype of the exact type of the other.

6.4 Commands as constructors

Potentially, the interpretation of commands as constructors could lead to inefficient use of storage. In general, for an assignment of the form $x := A$ it would be necessary to create the value returned by A in a new area of storage and then copy that value into the storage allotted for x. We have observed (section 5.2) that for non-creation command f, $x.f(\ldots)$ is equivalent to writing $x := x.f(\ldots)$, for x is of value type, and for creation command g, $x.g(\ldots)$ is equivalent to writing $x := g(\ldots)$. Further, we have argued (section 4.4) that the existing operational semantics is compatible with this. Thus the use of storage in performing assignments of either of the above forms can be as efficient as any produced by existing compilers when interpreting the corresponding calls. We suspect that optimisations would include the direct manipulation of storage allotted for x, rather than working on a copy.

6.5 Compatibility with existing operational semantics

The main changes in the operational semantics proposed here are:

1. semi-deep equivalence of values, instead of simple attribute equivalence

2. semi-deep copying of values, instead of simple attribute copying

3. the use of commands as constructors

(3) is fully compatible with the existing semantics. We have just allowed commands to be used in expressions, observing that the existing semantics of value types effectively treats commands as value returning queries.

With regard to (1) and (2), if an existing program uses the fact that simple attribute equivalence/copying is used to effect value equivalence/assignment, then there may be problems. However, we suspect that this will be the case in only a handful of situations if at all.

7 Summary

The paper has advocated the use of value types in OO programming, specifictaion and design, in general, and in Eiffel, in particular. It has identified problems with the modelling of value types in Eiffel using expanded types: problems with defining constructors; no value sub-typing; problems with implementation of value types. A small syntacic extension and an abstract semantics has been given which resolves these problems: commands are interpreted as constructors and transformers; sub-typing is admitted; query equivalence, instead of attribute equivalence, allows greater flexibility in implementating value types. An operational semantics, which is closer in spirit to the abstract semantics than Meyer's semantics for expanded types [Meyer, 1992], has been outlined. This retains most of the advantages of the abstract semantics. The compatibility of the extended language and revised semantics with existing Eiffel has been discussed.

8 Related Work

As far as we are aware, no research on value types in Eiffel has been reported, apart from the original work on expanded types by Meyer [Meyer, 1988, Meyer, 1992]. The work is clearly related to work in formal methods, in particular the extension of specification languages such as Z, VDM and OBJ with OO constructs [Lano and Haughton, 1994]. All these support the use and definition of value types, though some better than others. The difference with our work is that it brings formal specification and programming under the one seamless, umbrella, namely Eiffel. This is perhaps closer in spirit to the Larch method [Guttag *et al.*, 1985], which attempts to bridge the gap between specification and programming by using a *shared* language for defining the value types, and an *interface* language (one for each programming language) which acts as a specification language for specifying programs, where the assertions (pre, post conditions) on routines are essentially written in terms of the shared language. The main difference with our approach is that the shared language becomes part of the programming language itself, which allows value types to be used directly in a program. A possible consequence of this, as discussed below, is the automatic generation of reference types from value types.

9 Further Work

Clearly work needs to be done to implement the ideas described here in Eiffel. This task falls to the compiler writer, who would need to encode the operational semantics outlined in section 6. Fortunately, as we have shown, this semantics is largely compatible with the existing semantics, so such a rewrite should not significantly affect existing code. In so doing, it would, in our view, be sensible to change the ***expanded*** keyword to ***value*** to reflect the fact that expanded types are really value types. The changes to allowable creation routines in expanded types, proposed in section 5, including the addition of a ***default*** keyword, would also need to be incorporated. These extensions are compatible with the existing language, so should be compatible with already written code.

One immediate aim is to apply the ideas to the

```
expanded class STACK[X] creation
    new

feature -- queries
    top: X

    empty: BOOLEAN

feature -- commands
    new
        ensure
            empty;

    push(x: X)
        ensure
            top = x;
            pop = old Current;
            not empty;

    pop

end -- class STACK[X]
```

Figure 4: The stack remodelled

specification, implementation and general reorganisation of the basic types and data structure libraries for Eiffel. To see how this might be done, we return once again to the stack example.

In section 3, we identified a problem with the implementation of an expanded type, namely that one could not usefully use a reference type. The implementation of an expanded stack via an array was used as illustration. The problem is resolved in our semantics by weakening the semantics of expanded types. In section 5, we showed how our semantics allows constructors and transformers in the ADT definition of a value type to be modelled using commands in an Eiffel class, for the special case when that class is expanded. In this framework, a stack may now be remodelled as in figure 4.

The difference between this and figure 3 is that the transformers *push* and *pop* have been modelled as commands. Note that: the *creation* command *new* corresponds to the only constructor for stacks; *top* may be used in the ensure

condition of *push* due to our semantics of commands; the ensure condition on *push* could have been written as an invariant, which would have brought it closer in appearance to the usual ADT specification.

Now comes the interesting step. If one wished to transform this value type definition to a reference type, then one could do so by defining a new class say $REF\_STACK[X]$, which would have a single attribute of type $STACK[X]$. Furthermore, the queries and commands (including creation commands) of the new class would exactly match our remodelling of *STACK* in figure 4, where the queries would invoke their counterparts of the value attribute; similarly for commands, which would have the effect of changing the value held by the attribute as required. This process seems to be prescriptive enough to be automatic. If so, we would propose to incorporate a new keyword in Eiffel, namely ***reference*** to complement ***expanded***.

This still needs to be fully worked out and tested, and applied to more sophisticated data types.

The work described here also forms part of a general program of research which uses Eiffel as a platform for bringing the perceived benefits of formal specification and development techniques to the practising software engineer. This strand has resulted in the extension of Eiffel with quantification [Kent and Maung, 1995b] and for modelling value types (this paper). Another strand has considered how more run-time support can be given to checking of contracts as they currently exist in Eiffel [Mitchell *et al.*, 1995]. We have also begun to look at how higher level design concepts could be brought into the language [Kent and Maung, 1995a]. Attention will now focus on the provision of tool support (for example, how do our proposals affect contract checking), and on furthering the import of concepts from design methods, such as Syntropy [Cook and Daniels, 1994].

References

[Cook and Daniels, 1994] S. Cook and J. Daniels. *Designing Object Systems.* The Object-Oriented Series. Prentice Hall, 1994.

[Costa *et al.*, 1990] M.C. Costa, J. Cunningham, and J.P. Booth. Logical Animation. In *Proceedings of the International Conference on Software Engineering*, 1990.

[Guttag *et al.*, 1985] J.V. Guttag, J.J. Horning, and J.M. Wing. The Larch Family of Specification Languages. *IEEE Software*, 2(5):24–36, 1985.

[Jones *et al.*, 1991] C. B. Jones, K.D. Jones, P.A. Lindsay, and R. Moore. *Mural: A Formal Development Support System.* Springer Verlag, 1991.

[Kent and Maung, 1995a] S. Kent and I. Maung. Object Ownership and Aggregation. In *Proceedings of TOOLS Pacific 95.* Prentice Hall, 1995.

[Kent and Maung, 1995b] S. Kent and I. Maung. Quantified Assertions in Eiffel. In *Proceedings of TOOLS Pacific 95.* Prentice Hall, 1995.

[Lano and Haughton, 1994] K. Lano and H. Haughton, editors. *Object-Oriented Specification Case Studies.* The Object-Oriented Series. Prentice Hall, 1994.

[McKim and Mondou, 1993] J. McKim and D. Mondou. Class Interface Design: Designing for Correctness. *Journal of Systems and Software*, 23(2):85–92, 1993.

[Meyer, 1988] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988.

[Meyer, 1992] B. Meyer. *Eiffel: The Language.* The Object-Oriented Series. Prentice Hall, 1992.

[Mitchell *et al.*, 1995] R. Mitchell, I. Maung, J. Howse, and T. Heathcote. Checking Software Contracts. In *Proceedings of TOOLS USA 95.* Prentice Hall, 1995.

[Waldén and Nerson, 1994] K. Waldén and J. Nerson. *Seamless Object-Oriented Software Architecture: Analysis and Design of Reliable Systems.* The Object-Oriented Series. Prentice Hall, 1994.