To be cited as: Bertrand Meyer, *Towards practical proofs of class correctness*, to appear in in *||ZB 2003: Formal Specification and Development in Z and B*, Proceedings of 3rd International Conference, Turku, Finland, June 2003, eds. Didier Bert, Jonathan P. Bowen, Steve King and Marina Waldén, Lecture Notes in Computer Science 2651, Springer-Verlag, 2003, pages 359-387.

Towards practical proofs of class correctness

Bertrand Meyer

ABSTRACT

Preliminary steps towards a theory, framework and process for proving that contract-equipped classes satisfy their contracts, including when the run-time structure involves pointers; and its application to correctness proofs of routines from a *LINKED_LIST* class, such as element removal and list reversal.

1 SCOPE

“Trusted Components” are reusable software elements with guaranteed quality properties. Establishing a base of trusted components is among the most promising approaches to improving the general state of software; the potential for widespread reuse justifies the effort necessary to get the components right [11].

The most ambitious guarantee of component properties is a mathematical proof. The present work is part of an effort to produce a library of object-oriented components, equipped with contracts in the Eiffel style, and accompanied with mathematical proofs — mechanically checked — that the implementations satisfy the contracts.

We introduce a theory for correctness proofs of classes, and apply it to proofs for a class describing linked lists.

Like any realistic example of object-oriented component, the linked list class produces run-time structures relying extensively on pointers. A related set of articles [13] propose a general proof framework for pointers. The present article uses their results, but can be read independently. For more details about pointer semantics please refer to the complete series.

The scope of the Trusted Components effort is, of course, much broader than the work reported here. A Component Quality Model, under development, addresses the evaluation of commercial grade components from various technologies other than O-O classes, for example .NET assemblies and Enterprise Java Beans. The present discussion focuses on a special case: proving the correctness of classes. It is not the fulll story, but it's an important story — or, in the current state of this work, the beginning of an important story — that a Trusted Components project cannot afford to skip.

2 GUIDELINES

This work is based on some distinctive decisions.

We focus on the *object* structure. In descriptions of object technology developing the method's contribution to software engineering [10], the emphasis is naturally on *classes*, the compile-time module and type unit mechanism. Many formal treatments of object-oriented programming, such as Abadi and Cardelli's *Theory of Objects* [1], accordingly start from classes. To study the semantics of O-O computation, it seems more productive to start by modeling the run-time object structure, and the associated operations such as feature call, then work our way up — in a second step of the effort, only sketched in the present article — to classes and other program-level mechanisms such as inheritance.

In our study of these run-time object structures we'll take it for granted that they may include *pointers* (also called "references"). Although this is true of all realistic O-O programs and libraries, pointers have not been at the center of O-O theories; Abadi and Cardelli largely ignore them. By using high-level functions and associated operators we can model pointers in a simple way.

A useful literature exists on the formal treatment of pointers [3] [4] [8][15] [16] [17]. Some of it discusses this problem in a general context, whereas we will restrict the analysis to object-oriented programs. This means in particular that unlike many authors we won't concern ourselves with general pointer assignments *object*.*pointer_field* := *value* which, although still supported by recent languages from C++ and Java to C#, conflict with data abstraction principles. In O-O development one obtains the desired effect through a procedure call *object*.*set_pointer_field* (*value*) where *set_pointer_field* is a procedure of the corresponding class. Then the only legal form of assignment, and the only one we consider, is *field* := *value*, relative to the current object.

This notion of *current object* (*Current* in Eiffel, *self* or *this* in other languages) is central to the O-O method and to the model below. One of the most potent contributions of Simula 67, it is comparable in its depth to the notion of recursion in general programming. *Current* makes every operation relative: any variable, expression or operation is meaningful only in relation to the current object, which varies between successive executions of the same construct. To model this notion we will consider that any mathematical interpretation of a programming construct is a function whose single argument represents the current object. (The result of such a function is usually itself a function, representing for example a state transition.)

This reflects the actual behavior of object-oriented computation. In Eiffel the rule is explicit: executing a program is defined as creating one object, the *root*, and applying to it a specified *root procedure*. Upon execution, the root assumes the role of current object. If the procedure contains a call $x \bullet proc$ for some x of type C in the root's class, executing this call really means executing $root \bullet x \bullet proc$. If $proc$ itself contains an assignment $field := value$, both $field$ and $value$ must also be interpreted relatively, as denoting $root \bullet x \bullet field$ and $root \bullet x \bullet value$. This goes on: if $proc$ contains a call $y \bullet other\_proc$, any operation in op in $other\_proc$ really means $root \bullet x \bullet y \bullet op$ etc.

Even though the execution of any O-O programming construct is relative to the root-originated chain $root \bullet x \bullet y \bullet ...$ which determines its run-time target, the text of the construct, in the class where it appears, cannot know that target. To account for this fundamental property, any theory of object-oriented computation must be a “general relativity” theory.

Another characteristic of this work is that its specification techniques do not hesitate to take advantage of *high-level functions* and operations such as composition; assertions using these mechanisms will figure prominently in contracts, giving a power of expression that seems hard to match through other means such as first-order predicate calculus. Perhaps the most visible effect of this approach is that we'll be able to model the fundamental operation of object-oriented computation, a feature call $x \bullet f\,(a)$, through the mathematical expression $\overline{x} \bullet \overline{f}\,(\overline{a})$ where $\overline{x}$ and $\overline{f}$ are mathematical functions directly modeling x and f, $\overline{a}$ models the argument a, and “$\bullet$” is function composition.

The functions are *possibly partial* (abbreviated from now on to just “partial” if there is no ambiguity). Although many authors stay away from them, partial functions address many issues elegantly. For example we don't need any special concept to describe a void (null) pointer; it's simply a function applied outside of its domain. The reason for the common distrust of partial functions is the need to guard every function application $f\,(a)$ by a proof that a belongs to $\mathbf{domain}\,(f)$. We dodge this by almost never applying a function directly to its arguments, such as *composition*.

A final characteristic is the role of *models*. We can only prove a class correct relative to some view of the intended behavior of its instances. Rather than relying on a pure algebraic approach, we'll define such views through a mathematical model for the instances, interpreting for example a list as a sequence. Then we can specify the effect of a routine as its mathematical effect on the model. Combined with the use of high-level functions and operators, this gives us all the expressiveness we need. We will see that this technique has important practical consequences on the proof process: a key part will be the building of an appropriate model for the structures under study.

3 WHAT TO PROVE

Given classes that implement certain structures and associated operations, with contracts that specify the intended effect of the operations, the goal of the present effort is to prove that the implementations satisfy the contracts.

To understand the issues, let us start with an informal look at such a class and the kind of properties that will have to be proved.

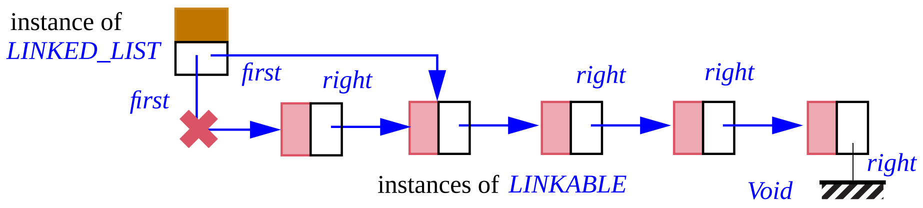
A routine

An example from the EiffelBase library is the following routine from the class *LINKED_LIST* [*G*]:

```
remove_front is
            -- Remove first element of list.
      require
            not_empty: first /= Void
      do
            first := first.right
      ensure
            one_less: count = old count – 1
            ... Other postcondition clauses (see text) ...
      end
```

Feature first is called first_element is the actual EiffelBase class.

The figure illustrates the underlying structure and the operation's effect. Class *LINKABLE* [*G*], complementing *LINKED_LIST*, describes individual list cells, each with a reference field *right* leading to the next cell if any.



Removing the front element

A property of such structures, which all public operations such as *remove_front* must maintain, is the absence of cycles; more precisely, starting from a *LINKED_LIST* instance and following *right* links zero or more times, we must never encounter a *LINKABLE* element twice, and end with a *Void*. The place to express such properties is the class invariant.

We must prove that, whenever the precondition (**require** clause) and the invariant both hold, executing the body (**do**) will lead to a state in which the postcondition (**ensure**) holds and the invariant holds again. Such proofs require a precise semantics for both the instructions and the assertions, as developed in the remaining sections.

Defining a model

Besides a semantic theory, we will need *models* of the object structures.

The example highlights the issue: contract expressiveness. The postcondition of *remove_front* states that the routine must decrease *count*, the number of list elements, by one, but omits the key property that the remaining elements are the same as before, except for the first, in their original order.

Contrary to a commonly encountered view, the solution does not have to involve extending the assertion language with first-order predicate calculus, which would be inadequate anyway to state many properties of interest. An example where predicate calculus doesn't appear to help is the invariant identified earlier: the absence of cycles.

It is more effective to focus on the abstract structure that an implementation class such as *LINKED_LIST* represents, and on the effect that operations have on it. In other words we introduce a model, expressing the *abstraction function* [6] associated with the class. For *LINKED_LIST* [*G*] the model should be a sequence of values (each of type *G*, the formal generic parameter). So if we assume the corresponding type *SEQUENCE* [*G*] we will have, in the class, a feature

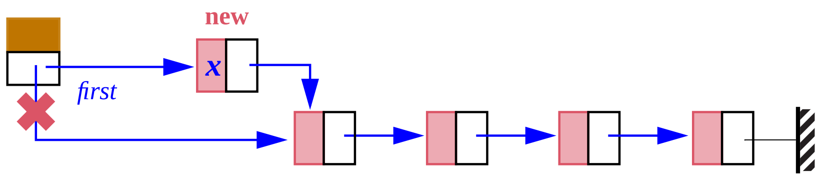
```
model: SEQUENCE [G]
            -- The sequence of values associated with this linked list
```

used for specification and proof purposes only. Another article [14] discusses in detail the use of such model features, showing in particular how to combine this notion with inheritance: if *LINKED_LIST* is just one of the descendants of a more general class *LIST*, whose other descendants such as *ARRAYED_LIST* provide alternative implementations, the *model* may be introduced in *LIST*, and [14] shows how to discharge much of the proof work in that higher-level class, so that the descendants only require a proof of implementation consistency. Here we limit ourselves to a simpler framework and do all the work in *LINKED_LIST*, ignoring inheritance. The immediate consequence is that we may now specify *remove_front* fully through the new postcondition

```
ensure
      head_chopped_off: model = old model.tail
```

where *tail* is a function on sequences, with the obvious meaning. Then we don't any more need the clause *one_less* (stating that *count* goes down by one) except as a theorem that will follow from the new clause *head_chopped_off* and the property of sequences that *s*.*tail*.*count* = *s*.*count* – *1*.

It is easy to apply the same approach to a routine *put_front* (*x*: *G*) that inserts an element at the beginnning, as illustrated:



Inserting at the front

The postcondition in this case is

```
extended: model = <x> + old model
```

where + denotes sequence concatenation and <x> a singleton sequence.

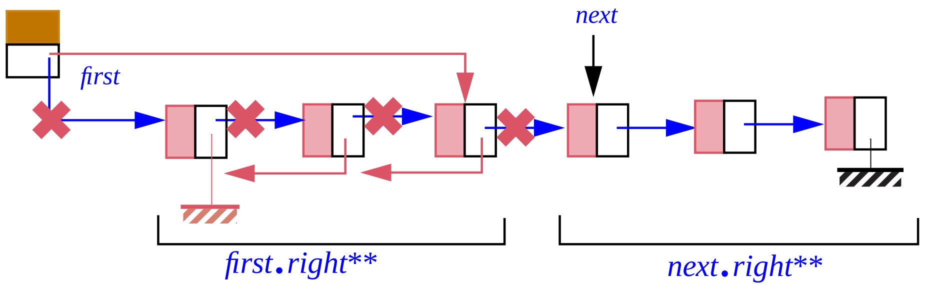
Once we have given ourselves a few more operations on sequences, we will also be able to express invariant properties such as the absence of cycles.

Reversing a list

The use of a model enables us to specify sophisticated operations, such as this list reversal routine using the procedure *put_right* from *LINKABLE*, which sets the *right* link. (The following figure helps understand the invariant.)

```
reverse is
        -- Change list to have the same elements in reverse order.
    local
        previous, next: LINKABLE [G]
    do
        from
            next := first
        invariant            -- See figure
            spliced: old model = [first.right**] □ mirror
                                + next.right**
        variant
            next.right**.count
        until
            next = Void
        loop
            [previous, first, next] := [first, next, next.right]
            first.put_right (previous)
        end
    ensure
        reversed: model = old model  □ mirror
    end
```

where *s* ▫ *mirror* is the mirror image of a sequence *s* and *f***, for a partial function *f*, is the function that for any *obj* yields the sequence *obj*, *f* (*obj*), *f* (*f* (*obj*)) etc., going for as long as defined. This is a generalization of reflexive transitive closure, hence the notation.



List reversing*: *intermediate state

To prove that the routine ensures its postcondition, it will suffice to prove that the loop body preserves the invariant, since on exit that invariant implies [*first*.*right***] ▫ *mirror* = **old** *model*, and *model* will be defined as *first*.*right***. The proof appears in section 10.

For brevity the invariant uses = instead of the object equality function *equal*. As in a postcondition, **old** *v* in the loop invariant denotes *v*'s value on entry to the routine. [*a*, *b*, ...] := [*x*, *y*, ...] denotes multiple simultaneous assignment.

The specification techniques illustrated by these examples take advantage of:

- The notion of model.
- To define models, any well-defined mathematical concept: here sequences, elsewhere sets, functions, relations, graphs etc.
- High-level operators on these structures, such as +, *mirror* and **.

This appears to give us the modeling power that we need to express the specifications of all practiacally useful data structures.

It remains to devise the semantic description techniques that will enable us to prove that the implementations satisfy these specifications.

4 NOTATIONS

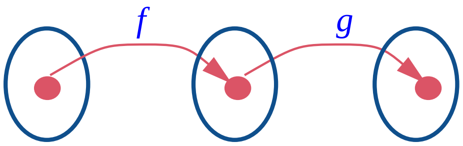
The following notations help keep the semantics and proofs simple.

Function abstraction

function *a* | *expression* denotes a function *f* such that *f* (*a*) = *expression* for any applicable *a*. This is plain lambda notation with keyword syntax. Although the approach is strongly typed we'll leave the type of *a* implicit.

Basic composition

The semantic models rely throughout on composition of relations and functions. Function composition will rely on the operator "$\square$", used in the loop invariant and postcondition of *reverse*; $f \square g$ is the **function** $a \mid g(f(a))$. The operands appear in the order of application:



Composition

The symbol differs slightly from the commonly used "$\circ$" to avoid any confusion, since the order of operands is reversed: $f \square g$ is the same as $g \circ f$.

Grouping and function application

Ordinary mathematical notation uses parentheses both for grouping and for function application; this can cause confusion when the elements grouped are themselves functions. For that reason, we reserve parentheses for function application, as in $f(a)$, and use brackets for grouping, as in

$$\boxed{[f \square g] \square h \;=\; f \square [g \square h]}$$

which, true for any f, g and h, expresses associativity of composition; applied to an individual element a this gives

$$\boxed{[[f \square g] \square h]\;(a) \;=\; [f \square [g \square h]]\;(a)}$$

using both grouping (brackets) and function application (parentheses). Associativity lets us omit many brackets, as in $f \square g \square h$.

> Some functional formalisms write function application simply by juxtaposing the function and the argument, as in $f\,a$. This convention has not been retained here as it would cause confusion. Since our functions are partial, we will anyway, as noted in section 2, use function application as little as possible.

Partial functions

$A \nrightarrow B$ is the set of partial functions from A to B; composition, defined for arbitrary relations, works well with partial functions.

When defining multi-level function spaces such as $A \nrightarrow [B \nrightarrow C]$, we may omit brackets associating from the right, writing $A \nrightarrow B \nrightarrow C$ in this example.

Rightmost composition

We will use two variants of composition, which are fundamentally the same operation as "$\square$" but with different signatures, made necessary by the multi-level function spaces involved in the semantic models.

The first variant is "rightmost composition". Consider f in $A \nrightarrow [B \nrightarrow C]$, so that that $f\,(a)$ for any applicable a is itself a function. Given a function g similarly in $A \nrightarrow [C \nrightarrow D]$, we can't use the ordinary composition $f \square g$ (the signatures don't match), but we may want to compose $f\,(a)$ and $g\,(a)$ for a given a. The resulting function will be written $f \blacksquare g$. This can go over several levels, with functions in $A_1 \nrightarrow \ldots A_n \nrightarrow X \nrightarrow Y$ and $A_1 \nrightarrow \ldots A_n \nrightarrow Y \nrightarrow Z$, for some sets X, Y and Z; the general definition is then:

$$\boxed{\begin{aligned} f \blacksquare g \;=\; &\textbf{function } a_1 \mid [\, \textbf{function } a_2 \mid [\ldots \mid [\textbf{function } a_n \mid \\ &[[\ldots[[\, f\,(a_1)]\,(a_2)]\ldots]\,(a_n)] \;\square\; [[\ldots[[g\,(a_1)](a_2)]\ldots]\,(a_n)]\,]\ldots] \end{aligned}}$$

We may similarly define the "rightmost inverse" f^{-1*} of such a function f as

$$\boxed{\textbf{function } a_1 \mid [\ldots \mid [\textbf{function } a_n \mid \quad [\ldots[\, f\,(a_1)]\ldots]\,(a_n)]^{-1}\,]\ldots]}$$

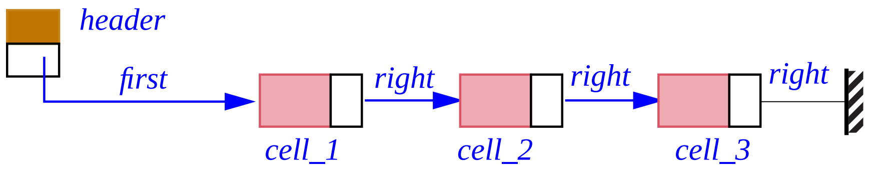
and, when the rightmost target set is $\mathbb{B}$ (booleans), the "rightmost implication" $f \overset{*}{\Rightarrow} g$ as

$$\boxed{\forall\, a_1, a_2, \ldots a_n \mid [\ldots[[\, f\,(a_1)]\,(a_2)]\ldots]\,(a_n)] \;\Rightarrow\; [\ldots[[g\,(a_1)](a_2)]\ldots]\,(a_n)]}$$

as well as the "rightmost conjunction" $\overset{*}{\wedge}$.

State-curried composition

The other variant of composition arises from the specific nature of our semantic functions which (as seen in the next sections) all have signatures of the form $A_1 \nrightarrow \ldots A_n \nrightarrow States \nrightarrow Y$, where $States$ is the set of possible run-time states. We will use such functions to model a linked list structure



Denoting successive list cells

by representing *first* and *right* as functions in $Objects \nrightarrow States \nrightarrow Objects$; given an object *header*, for example, $[first\ (header)]\ (s)$ is the object to which its *first* link points in state s — *cell_1* on the figure. It's desirable to compose such functions *applied in the same state*, for example *first* and *right* so that the result, applied to *header*, gives us the object labeled *cell_2*. This is the kind of expressiveness we need to state properties of the current state, in particular class invariants, loop invariants and other assertions.

We cannot directly compose two such functions, f in $X \nrightarrow States \nrightarrow Y$ and g in $Y \nrightarrow States \nrightarrow Z$. Fixing the state, however, we may compose their variants

$$\boxed{\begin{array}{l}\textbf{function}\ x \mid\ \ [f\ (x)]\ (s) \\ \textbf{function}\ x \mid\ \ [g\ (x)]\ (s)\end{array}}$$

for a given same state s, the same in both cases. This operation will simply use a period "$\bullet$". The definition of $f \bullet g$ is

$$\boxed{\textbf{function}\ x \mid\ \ [\textbf{function}\ s \mid\ \ [g\ ([f\ (x)]\ (s))]\ (s)\ ]}$$

This choice of symbol works well for modeling the object structures created by object-oriented programs: as the previous example indicates, the successive items of a list will be given by the functions *first*, $first \bullet right$, $first \bullet right \bullet right$ etc. all applied in the same state. It is indeed one of the results of this article that we can understand *feature application*, the central operation of object-oriented programming, as the mathematical notion of *function composition*. In the case of an attribute, the composition operator is "$\bullet$" as just seen; in the case of a routine it will be rightmost composition, "$\blacksquare$".

Both of these operators are fundamentally the same as composition; they simply massage the order of arguments to remove any signature mismatch. The purpose is clear: express as many properties as possible through composition operators. On first reading of the following discussion you may disregard the differences between "$\bullet$", "$\square$" and "$\blacksquare$", just seeing them as composition tuned to the required signature in each case. All the examples of this discussion and other proofs based on the theory must be mechanically type-checked, to ensure use of the proper variant in each case and justify this call for the reader's trust.

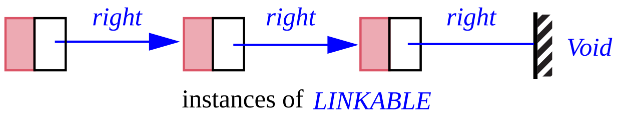
5 BASIC MODELING ASSUMPTIONS

The state

The set of possible state is called *States*. An element of *States* describes the instantaneous state of an object-oriented program's execution, and is defined by a set *Objects* and a collection of functions:

- *Objects* denotes of set of addresses hosting objects; it's a subset of *Addresses*, the set of possible (abstract) memory addresses [13]. Note that an element of *Objects* doesn't represent the contents of an object, but its location; this reflects the notion of *object identity*, making it possible to consider each object individually regardless of its contents.
- There's also a collection of functions in $Objects \nrightarrow Objects$ (the set of partial functions from *Objects* to itself) representing objects' reference fields. Class *LINKABLE*, for example, has an attribute *right*: *LINKABLE* [*G*] as illustrated next, which yields a function, also called *right*, in the model. When discussing the properties of such functions in general, we'll give them names like *x*, *y*, ...

"Right" links in LINKABLE objects



- Objects may also have non-reference fields; for example class *LINKABLE* [*G*] has an attribute *item*: *G* representing the values in cells, shown as the shaded areas in the figure. For cells of type *LINKABLE* [*INTEGER*], *item* fields denote integers. Such attributes are represented by functions in $Objects \nrightarrow Expanded$, where *Expanded* is the set of possible non-reference values including booleans, integers, real numbers etc. General names for such functions are *u*, *v*, ...

Values will denote the set of all possible values: $Objects \cup Expanded$.

All the functions involved are partial (meaning, by the earlier convention, *possibly* partial). Partiality helps us in two different ways:

- It gives us a simple interpretation for *Void*: we model a void reference, such as the rightmost link on the last figure, simply by ensuring that the function, here *right*, is not defined for the corresponding object.
- We can also use partial functions to handle type rules of a statically typed O-O language, by defining a function such as *right* so that its domain is a subset of the set of *LINKABLE* objects.

Other conventions

Names such as f, g, ... denote functions from *Objects* to either *Objects* or *Expanded*, representing fields that contain either references or other values. Names such as *obj*, *obj1*... denote objects; a, b, ... denote objects or values.

For elements of the software text: i, j, ... denote instructions; r, s, ... denote routines. All our routine calls will have exactly one argument, as in $r\ (a)$ or $x \bullet r\ (a)$; this causes no loss of generality if we assume that the set of values, as in Eiffel, includes a *TUPLE* type.

Assertions, for which we will use names such as P, Q, ..., denote boolean properties applicable to a certain object in a certain state. For example the assertion $n > 0$ is a function of the state, true in states for which n, evaluated on the current object, is positive. Accordingly, class invariants will be modeled as functions in $Objects \nrightarrow States \nrightarrow \mathbb{B}$, and pre- or postconditions of a routine with arguments as functions in $Values \nrightarrow Objects \nrightarrow States \nrightarrow \mathbb{B}$.

Interpreting state changes

The execution of an object-oriented program consists of a sequence of state changes reflecting execution of individual constructs, for example a procedure call $x \bullet r\ (a)$.

The basic semantics defined below is of the *denotational* style; this means we define the meaning of a typical imperative construct as a function in $A_1 \nrightarrow ...A_n \nrightarrow States \nrightarrow States$ where A_1, ... A_n hold the parameters of the construct, and the resulting $States \nrightarrow States$ function describes the new state produced by the construct in terms of the previous state. This approach is mathematically simplest.

When applying the specification through a proof workbench such as Atelier B [2] [5] we may take advantage of a predefined notion of event covering the $States \nrightarrow States$ transformations in a more *operational* style, making the functions implicit. Since we are interested in pre-post properties of routines, proofs will use a partly *axiomatic* style.

We may compose state transformations. In the denotational view this operation uses rightmost composition as defined in section 4; in a more *operational* interpretation it simply means executing events in sequence. The formal properties are in direct correspondence.

Possible state changes

The state being defined by a set of objects and a collection of functions on these objects, an elementary state change will either

1 • Change the set of *Objects* by adding or removing an object.

2 • Change one of the functions; at the most basic level this means changing the value of one of the functions on one of its possible arguments.

There is no such thing as "changing an object" in this model. The model for an object is just an integer, representing its abstract "address". So we may add or remove an object (events of type 1), but to model the changing of a field in an existing object — what an O-O programmer would think of as changing the object — we use events of type 2, changing the corresponding function. Assume for example that class *EMPLOYEE* has an attribute *age*: *INTEGER*. If we execute *Jill*.*pass_birthday* where *pass_birthday* is a procedure of the class that performs

```
age := age + 1
```

the mathematical effect is to change function *age* so that its value for that object, *age* (*Jill*), is increased by one.

Events of type 1 correspond to object allocation (by the program) and deallocation (by the program or a garbage collector). They are studied in detail in the second part ("coarse-grain model") of [13]. For the present discussion we don't consider them: the set of objects is fixed, and all that happens is procedure calls that modify these objects — meaning, as we have just seen, changing some values of the applicable functions. The case of a procedure that may create an object will be handled by combining the two discussions.

Function substitution

The basic operation, representing an event of type 2, modifies the value of a function for a single object. Given two functions *f* and *g*, the expression

```
f := g
```

denotes the function in $Objects \nrightarrow [States \nrightarrow States]$ that informally yields, for any object *obj*, the state transformation that changes nothing except the value of $f(obj)$, which will now be $g(obj)$ if defined, undefined otherwise (the functions involved are partial).

Here is a more formal definition of this operation. We saw that the state has two components: the set *Objects* representing object identities, and a set of functions in $Objects ⇸ Objects$ or $Objects ⇸ Values$, each with a name such as f, g, h... We may denote these functions, for a given state s, as $s{\bullet}f$, $s{\bullet}g$, ... Then $f := g$ is a function in $Objects ⇸ [States ⇸ States]$; call it *assign*. For any object *obj* and state s, $assign\ (obj)$ is a function from *States* to *States*, so $[assign\ (obj)]\ (s)$ is a state; call it s'. Then s' is the same state as s except for its f component, the function we're calling $s'{\bullet}f$. That function is the same as $s{\bullet}f$ except at *obj*:

- $s'{\bullet}f\ (obj') = s{\bullet}f\ (obj')$ -- For *obj'* other than *obj*
- $s'{\bullet}f\ (obj) = s{\bullet}g\ (obj)$ -- Or undefined if *obj* is not in the domain of $s.g$.

The $:=$ operator enjoys two characteristic properties:

$$
\boxed{
\begin{array}{lll}
\text{[A1]} & [[f := g] \blacksquare f] = g & \\
\text{[A2]} & [[f := g] \blacksquare h] = h & \text{-- For a function } h \text{ other than } f
\end{array}
}
$$

Properties are numbered in a single sequence, with different initial letters, A *for axiom,* S *for semantics etc.*

where the use of composition (more precisely, rightmost composition $\blacksquare$) avoids having to worry about the functions being defined or not.

The $:=$ operator will, through these properties, enable simple proofs of attribute assignment instructions $x := y$ in a class text. An advantage is that unlike traditional assignment axioms the rules do not involve textual substitutions or other transformations of the program text; they simply rely on function composition.

Unlike the Hoare assignment axiom, these properties work forward; but their practical application in the examples that follow leads to a backward style similar to weakest precondition calculus.

Note the signature $Objects ⇸ [States ⇸ States]$ of $f := g$: the operation changes the state of a single function at a *single point*, denoted by the *Objects* argument. The B notation, for a specific *obj* in *Objects*, would be $f\ (obj) := g\ (obj)$; the repetition of *obj* explains why we need a special notation where *obj* appears just once. This special role of *obj* illuminates the special role of the “current object” in object-oriented computation.

We may generalize function substitution to multiple sources and targets:

$$
\boxed{[f_1, f_2, ...\ f_n] := [g_1, g_2, ...\ g_n]}
$$

with the corresponding generalization of the characteristic properties, reflecting that the substitutions are simultaneous:

$$
\boxed{
\begin{array}{lll}
\text{[A3]} & [[f_1, ...\ f_n := g_1, ...\ g_n] \blacksquare f_i] = g_i & \text{-- For } 1 \leq i \leq n \\
\text{[A4]} & [[f_1, ...\ f_n := g_1, ...\ g_n] \blacksquare h] = h & \text{-- For } h \text{ other than all } f_i
\end{array}
}
$$

6 THE SEMANTIC RULES

We are now ready to examine the semantics of the object-oriented mechanisms. The denotation of a programming language construct c — the mathematical model for c — will be written $\overline{c}$.

The following table, followed by an explanation of every entry, specifies the semantics of the core subset of an object-oriented language such as Eiffel. Each entry gives the denotation $\overline{c}$ of a different construct c; the last column gives the signature of that denotation, that is to say the set (of partial functions) to which it belongs. The last entry, [S19], gives the cumulative definition of the correctness of a routine having a pre- and postcondition.

Construct		**Denotation**	**Signature**
Basic constructs			
[S5] Name of attribute of class	$\overline{f}$	$= f$	*Objects* ⇸ *States* ⇸ *Values*
[S6] Attribute assignment	$\overline{f := g}$	$= \overline{f} := \overline{g}$	*Objects* ⇸ *States* ⇸ *States*
[S7] Multiple attribute assign-ment	$\overline{[f_1,...f_n] := [g_1,...g_n]}$	$= [\overline{f_1}, ... \overline{f_n}] := [\overline{g_1}, ... \overline{g_n}]$	*Objects* ⇸ *States* ⇸ *States*
[S8] Instruction sequencing	$\overline{i\ ;\ j}$	$= \overline{i} \blacksquare \overline{j}$	*Objects* ⇸ *States* ⇸ *States*
[S9] Attribute call	$\overline{x \bullet f}$	$= \overline{x} \bullet \overline{f}$	*Objects* ⇸ *States* ⇸ *Objects*
***Routine*: r (a) is require *pre* do *body* ensure *post* end**			
[S10] Routine (overall semantics)	$\overline{r}$	$=$ **function** $a \mid \overline{body}$	*Values* ⇸ *Objects* ⇸ *States* ⇸ *States*
[S11] Routine call, unqualified	$\overline{r\ (u)}$	$= \overline{r}\ (\overline{u})$	*Objects* ⇸ *States* ⇸ *States* -- For procedure *Objects* ⇸ *States* ⇸ *Values* -- For function
[S12] Routine call, qualified	$\overline{x \bullet r\ (u)}$	$= \overline{x} \blacksquare \overline{r}\ (\overline{u})$	Same as above entry [S11]
[S13] Unary expression (for operator §)	$\overline{§\ a}$	$= \overline{§}\ \overline{a}$ (Use [S11], [S12])	*Objects* ⇸ *States* ⇸ *Values*
[S14] Binary expression (for operator §)	$\overline{a\ §\ b}$	$= \overline{a}\ \overline{§}\ \overline{b}$ (Use [S11], [S12])	*Objects* ⇸ *States* ⇸ *Values*

Construct	**Denotation**	**Signature**
Assertions and correctness **(*in routine* $r\ (a)$)**		
[S15] Class or loop invariant Use [S11], [S12], [S13], [S14]		$Objects \nrightarrow States \nrightarrow \mathbb{B}$
[S16] Pre- or post-condition Use [S11], [S12], [S13], [S14]		$Values \nrightarrow Objects \nrightarrow States \nrightarrow \mathbb{B}$
[S17] Postcondition clause $\overline{\textbf{ensure}\ Q}$ =	$\overline{r} \blacksquare \overline{Q}$	$Values \nrightarrow Objects \nrightarrow States \nrightarrow \mathbb{B}$
[S18] "Old" equality in postcondition $\overline{\textbf{ensure}\ f = \textbf{old}\ g}$ =	$[\ \overline{r} \blacksquare \overline{f} = \overline{g}\ ]$	$Objects \nrightarrow States \nrightarrow \mathbb{B}$
[S19] Routine correctness	$[\ \overline{pre}\ \overset{*}{\curlywedge}\ \overline{inv}\ \overset{*}{\Rightarrow}$ $\overline{r} \blacksquare [\ \overline{post}\ \ \overline{inv}\ ]]$	$\mathbb{B}$

The left-hand sides of the equalities cite constructs of an object-oriented programming language; the right-hand sides are their mathematical denotations. The rightmost column gives the signatures of these denotations — the mathematical sets to which they belong (sets of functions in all cases).

In case [S5] we simply prescribe that for any attribute f of the class the model will include a corresponding function. The rule $\overline{f} = f$ indicates that we use the same name for the function in the model as for the attribute in the software text.

Case [S6], $f := g$, is a standard assignment. The fields represented by f and g may be value fields, for example of type *INTEGER*, or reference fields leading to other objects or *Void*. The mathematical intepretation of such an assignment as an operation of the state is that it replaces the value of f, for any object *obj* to which the assignment is applied, by the value of g for that object.

> The operator := on the left-hand side is assignment, from the programming language; := on the right side is a mathematical operator, function substitution. Inventing a new operator for the latter purpose would avoid the risk of confusion but make the notation more complex.

This rule, [S6], captures the essence of the "current object" in object technology. The key is that we do not specify any particular object: $\overline{f} := \overline{g}$ is a function, applicable to an object. Both the target and the source of the assignment are themselves functions applicable to an object; the effect of the assignment is to replace the value of the target function on that object, whatever it is, by the value of the source for the same object.

Since in this discussion we do not consider such operations as object creation, all run-time events ultimately reduce to operations such as $f := g$ whose model is a function from *Objects* to state transformers.

Case [S7] is the generalization to multiple sources and targets.

In case [S8] $i\ ;\ j$ is the instruction sequence that executes i then j. As we model instructions by functions, the model for their sequence is the composition of their models. We must use rightmost composition "$\blacksquare$" rather than ordinary composition "$\square$" since the denotations of i and j are not state transformers but functions from *Objects* to state transformers; recall that $i \blacksquare j$ ← *Page 10.*
is the function that, for any obj, yields $i\ (obj)$ composed with $j\ (obj)$.

Case [S9], "attribute call", applies an attribute to an object, and is written $x \bullet f$ in O-O notation; it is pleasant to model it through function composition as $\overline{x} \bullet \overline{f}$. Calls to the other kind of feature, routines, will have a similar rule [S12].

Case [S10] defines the semantics of a routine r as being, for any argument a, the semantics $\overline{body}$ of the routine's body as applied to a. The name r is not by itself a construct, but defining a semantics for r helps define the model of the actual constructs involving r: calls to the routine.

Case [S11], $r\ (u)$, is the first kind of such call: **unqualified**, that is to say, executed from a routine of the same class and using the current object as target. The effect is simply to apply the function $\overline{r}$, the semantics of r, to the denotation $\overline{u}$ of u.

Case [S12] is the second kind of call, **qualified**: $x \bullet r\ (u)$ applies a certain feature to a certain explicitly named target with certain arguments. The observation here is that x as well as u are, mathematically, functions on objects; so is r with an extra degree of abstraction corresponding to the argument. The mathematical equivalent is simply $\overline{x} \blacksquare \overline{r}\ (\overline{u})$, closely mirroring the programming language notation as in attribute call [S9]. This rule shows feature application, the fundamental computational mechanism of object-oriented development, as function composition and function application.

Cases [S13] and [S14] acknowledge the property that (in Eiffel at least) an expression involving a unary or binary operator is just an abbreviation for a function call; for example $a + b$ is formally a function call $a \bullet plus\ (b)$ where $plus$ is the function **infix** "+" associated with the operator. So to handle these cases we just apply the function's model to the operands' model. The same approach will work for predefined equality and inequality operators: the model for $\overline{a = b}$ is $\overline{a} = \overline{b}$.

This also gives the model for a class or loop invariant [S15] since it's a boolean-valued expression. The signature is $Objects \nrightarrow [States \nrightarrow \mathbb{B}]$: applied to any object, the model is a boolean-valued function of the state.

The same holds of a routine's precondition or postcondition [S16] with an extra function level — the initial *Values* in the signature — corresponding to the argument of the enclosing routine r.

It is convenient to give a model [S17] to a postcondition clause **ensure** Q, the composition of the semantics of r and the semantics of Q. The signature of $\overline{r} \blacksquare \overline{Q}$ is $Values \nrightarrow Objects \nrightarrow States \nrightarrow \mathbb{B}$ (this is also the signature of $\overline{Q}$, and the signature of $\overline{r}$ is $Values \nrightarrow Objects \nrightarrow States \nrightarrow States$).

In such a postcondition, we may encounter case [S18], a reference to **old** f where f is an expression, of signature $Objects \nrightarrow States \nrightarrow Values$ (f may not involve the routine's arguments). This represents the value of f evaluated on the current object on entry to the routine r. The mathematical model must "unwind" the semantics of r; rule [S18] addresses the common case of a postcondition clause of the form f = **old** g, which we interpret as $\overline{r} \bullet \overline{f} = \overline{g}$, expressing that the value of f in the state resulting from executing r is the original value of g.

These properties yield a practical strategy for dealing with **old**, illustrated by the example proofs of the following sections. To prove a routine correct, we compute $\overline{r} \blacksquare \overline{Q}$; as the body of r is usually a sequence of instructions $i; \ldots j; k$ this means, from the rule case [S8], computing $Q_1 = \overline{k} \blacksquare \overline{Q}$, then $Q_2 = \overline{k} \blacksquare Q_2$ and so on back to i. The practical rule for **old** is that we may keep any **old** x subexpression unchanged throughout this process; then when we get back to the beginning of the sequence, to prove per [S19] that the initial assumption implies the expression we have obtained, we drop the **old**.

Case [S19] gives the basic proof obligation for a routine: that the precondition and invariant imply the postcondition and invariant evaluated in the state resulting from executing the routine. A similar rule will apply, for example, to the proof that a loop body satisfies the loop invariant.

Viewed as definitions of the semantics, these equalities are recursive; for example the denotation of $f := g$ refers to $\overline{g}$, the denotation of g. Since the routines may themselves be recursive, the equalities do not actually provide a proper definition of the semantics unless we use a fixpoint interpretation. We can avoid the issue by noting that the goal is to prove properties of routines through [S19]; if encountering a recursive call, we will assume the property to prove, in line with Hoare's axiomatics of routines [7].

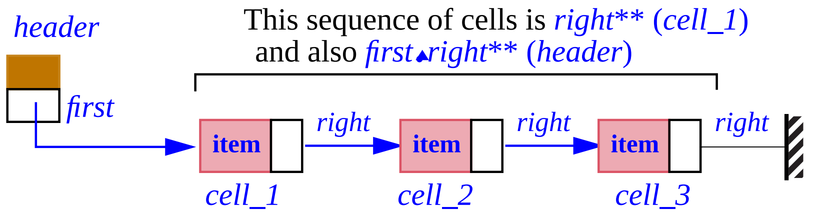
7 OPERATOR PROPERTIES

The proofs that follow will use some properties of the functional operators used in modeling object structures.

Sequence closure

Introduced earlier, "sequence closure", *f*** for a function *f*: *A* ⇸ *A*, is the function that for any *x*: *A* yields the sequence *x*, *f* (*x*), *f* (*f* (*x*)) ..., up to the first value that is outside of the domain of *f*. (The formal definition is not included but poses no difficulty.) For all the examples of this article the structures are acyclic so the sequence is finite.

In light of the preceding discussion of the "•" operator we may generalize the notation to a function *f* representing an attribute, and hence of signature *Objects* ⇸ *States* ⇸ *Objects* rather than just *Objects* ⇸ *Objects*: we just take *f***, for a given state *s*, as denoting the application of ** to **function** *obj* | [**function** *s* | [*f* (*obj*)] (*s*)]. Here is an example:


header
This sequence of cells is right** (cell_1)
and also first•right** (header)
first
item
right
item
right
item
right
cell_1
cell_2
cell_3

Reflexive-transitive sequence closure

In a given state, function *right*** applied to the first *LINKABLE* cell *cell_1*, yields the sequence of *LINKABLE* cells consisting of *cell_1*, *cell_2* and *cell_3*. Applying *first*•*right*** to the *header* object yields the same sequence.

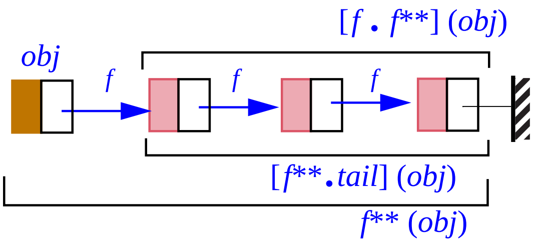
Sequence closure properties

The following properties involve functions *f*, *g*, ... in *Objects* ⇸ *States* ⇸ *Objects*. The state plays no explicit role — it is the same throughout — so the properties will also hold for functions in *Objects* ⇸ *Objects* if we replace "•" by plain composition "◦". By including *States* in the signature we cover the intended application to functions *f*, *g*, ... representing reference attributes.

The first property relates sequence closure and composition:

$$\boxed{\text{[T20]}\ f \bullet f^{**} = f^{**} \bullet tail}$$

In words — as illustrated below in the application of both sides to an argument *obj* in a given state —this expresses that if you start from an object and follow the *f* link once, the *f* sequence starting at the resulting object is the tail of the *f* sequence starting at the original object:



Tail and composition

Two corollaries are

[T21]	f^{**}	$= <f> + f \bullet f^{**}$
[T22]	$f \bullet g^{**}$	$= <f> + f \bullet g \bullet g^{**}$

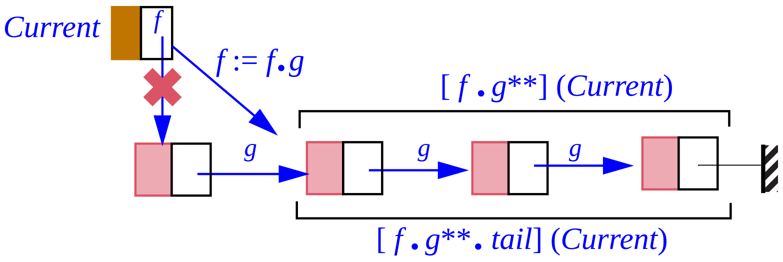
We may use the notation f^{++} for the expression $f \bullet f^{**}$ that appears in both [T20] and [T21]; it yields for any x the sequence $f(x), f(f(x))...$, that is to say $f^{**}(x)$ deprived of its first element. [T21] indicates that $f^{**} = <f> + f^{++}$.

Sequence closure and function substitution

A related property (involving the state) combines sequence closure and function substitution:

$$\text{[T23]}\quad [f := f \bullet g] \blacksquare [f \bullet g^{**}] \;=\; f \bullet g^{**} \bullet tail$$

which we may illustrate as follows:


Current
f
$f := f \bullet g$
$[f \bullet g^{**}]\ (Current)$
g
g
g
$[f \bullet g^{**} \bullet tail]\ (Current)$

Tail and composition

In words: consider the g sequence starting at the target of the f link from the current object. (It's $f \bullet g^{**}$, appearing on both sides of [T23].) Replacing the f link of *Current* by $f \bullet g$ implies replacing that sequence by its tail.

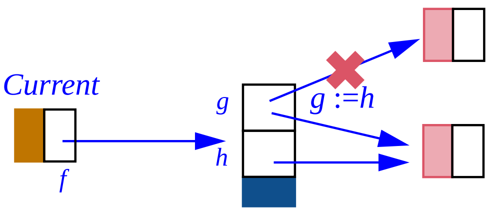
Proof of [T23]: the property [A1] of function substitution lets us simplify the left-hand side into

$$\boxed{f \bullet g \bullet g^{**}}$$

which, by applying [T20] to g, yields the right-hand side.

The next property, illustrated below, enables us to deal with the effect of remote assignments by deducing that after a call $f \bullet set\_g\ (h)$, where $set\_g\ (a)$ performs $g := a$, the value of $f \bullet g$ will be h:

$$\boxed{\text{[T24]}\quad [f \bullet [g := h]] \blacksquare [f \bullet g] \;=\; h}$$



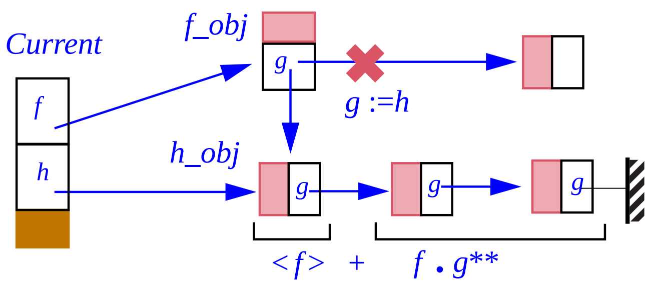
Effect of remote assignment

Proof: apply both sides to an object obj and let $obj' = f\ (obj)$. From the definition of $\blacksquare$ the left-hand side is $[[g := h]\ (obj')] \bullet [[f \bullet g]\ (obj)]$, that is, $[[g := h]\ (obj')] \bullet [g\ (obj')]$, which from the definition of function substitution [A1] is $h\ (obj')$.

As a consequence:

$$\boxed{\text{[T25]}\quad [f \bullet [g := h]] \blacksquare [f \bullet g^{**}] \;=\; <f> + h \bullet g^{**}}$$

In words (which you may want to follow on the next figure): call f_obj, as illustrated, the target of the f link from the *Current* object. The left side of [T25] denotes the g sequence from f_obj — the sequence $f \bullet g^{**}$ — evaluated in the state resulting from reattaching, in f_obj, the g link to the target h_obj of the h link. The right side is, in the original state, the sequence that starts with f_obj and continues with the g sequence beginning at h_obj.



Effect of remote assignment

Proof: from [T22] we write the left side as $[f \bullet [g := h]] \blacksquare [\langle f\rangle + f \bullet g \bullet g^{**}]$. Distributing $\blacksquare$ over the concatenation operator $+$ and applying [T24] gives the right side.

Finally, we will use the following elementary property, given without proof, of the *mirror* and concatenation operations on sequences:

[T26] $(s1 + s2) \; \square \; mirror = s2 \; \square \; mirror + s1 \; \square \; mirror$

8 MODELING LINKED LISTS

We will apply the preceding semantic rules to prove the correctness of *LINKED_LIST* routines *remove_front* and *reverse*. This requires expressing more precisely the properties of the model used for this class. The experience gained so far in proving properties of classes indicates that this step of devising a proper model is just as important as the task of performing the proofs once a model has been devised.

Sequences and their properties

As noted earlier, we associate with an instance of *LINKED_LIST* [*G*] a *model* of type *SEQUENCE* [*G*], representing the sequence of its values. The basic property of the *model* may be expressed as a class invariant, relative to the current state:

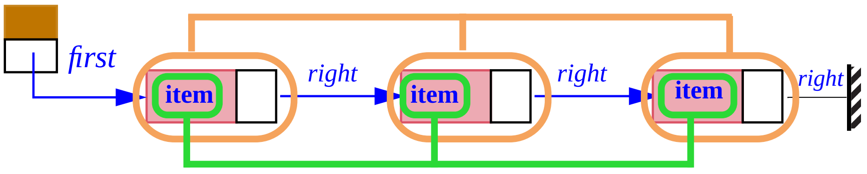
[A27] $model = first \bullet right^{**} \bullet item$

The decision to define a *SEQUENCE* model for every list object belongs in a higher-level class, *LIST*, of which *LINKED_LIST* is a descendant. The linked-list *model* is an implementation of the abstract *model* from *LIST*. This overall structure and the relation of the proof technique to inheritance are covered in [14]; in the present discussion we examine the implementation class independently of its ancestry.

As illustrated below, *first* **.** *right*** is the sequence of *LINKABLE* [*G*] cells making up the list; we could use it as the model, but what is of interest to users of the list is not the sequence of cells, it's the sequence of *G* values they host, which the definition [A27] gives us by composing *first* **.** *right*** with *item*.

This relies on the standard definition of a finite sequence *s* as a function from an integer interval starting at 1 to a set *X*, here *LINKABLE* [*G*]. If *f* is a function in $X \rightarrow G$ for some *G*, the composition *s* **.** *f* describes another sequence, with values in *G*, obtained by applying *f* to every element of *s*.

first **.** *right*** (sequence of *LINKABLE* [*G*] cells)



first **.** *right*** **.** *item* (sequence of *G* values)

Modeling a linked list as a sequence of values

The function *item* comes, like *right*, from class *LINKABLE*:

```
class LINKABLE [G] feature
        right: LINKABLE [G]
                -- Reference to next cell
        item: G
                -- Value stored in cell
        ... Routines (see below) ...
end
```

Like the above definition of *model*, all formulae of interest include the composition "**.** *item*" as their last element; as a result we can remove it in all equalities between such formulae. For brevity we will from now on ignore *item*, using for *model* the simplified version

[A28] *model* = *first* **.** *right***

as if we were dealing with a sequence of *LINKABLE* [*G*] items rather than a sequence of *G* values. This simplification was already present in the loop invariant of the *reverse* procedure (page 7).

A property of the model is:

[T29] [*first* := *first* **.** *right*] ■ *model* = *model* **.** *tail*

Proof: the left-hand side, through the definition [A27] of *model*, is

$$[\, first := first \bullet right \,] \;\blacksquare\; [\, first \bullet right^{**} \,]$$

The property [A1] of function substitution lets us simplify this into

$$first \bullet right \bullet right^{**}$$

The right-hand side, again from the definition [A27] of *model*, is

$$first \bullet right^{**} \bullet tail$$

yielding [T29] as a consequence of [T20].

Prohibiting cycles and tail sharing

Our linked list structures must be acyclic. This will give another invariant clause, which we may express as the requirement that, in any state *s*

$$\text{[A30]}\quad injective\ ([\, first \bullet right^{**} \,]\ (s))$$

where *injective* (*r*), for a relation *r* (including the case of a function) indicates that *r* never pairs two different source elements with the same target element; this can be defined as $r \,\square\, r^{-1} \subseteq Id$ where *Id* is the identity relation. [A30] states that a *right* sequence may not include the same *LINKABLE* [*G*] cell twice, although two of its cells may of course have the same *G* content.

We must also preclude tail sharing: no two lists may share *LINKABLE* [*G*] cells (although they may again share cell values). The invariant clause is

$$\text{[A31]}\quad injective\ ([\, first \bullet right^{*} \,]\ (\bullet\, s \,\bullet))$$

This is almost the same as [A30], using the reflexive transitive closure of *right* rather than sequence closure. Because this yields a relation, not a function, we need the image operator $(\bullet \ldots \bullet)$ [13] rather than function application.

The correctness rule [S19] requires every exported routine of the class to maintain [A30] and [A31].

9 PROVING CORRECTNESS OF LIST REMOVAL

Let's apply the theory to prove the procedure *remove_front* introduced earlier.

> It is in general meaningless to talk of "proving software": you prove the correctness of a software element not in the absolute but with respect to a certain specification. Our classes and their routines, however, are equipped with contracts, so "proving a routine" simply means proving that it satisfies its specification as expressed by the contract.

The postcondition, labeled *head_chopped_off* on page 6, is *model* = **old** *model.tail*. [S18] tells us that the property to prove is then

$$\boxed{\overline{remove\_front} \blacksquare\ model = model.tail}$$

From [S10], $\overline{remove\_front}$ is $\textbf{function}\ a \mid \overline{body}$ where *body* denotes the body of the procedure and we can ignore *a* since the procedure has no argument. Its body is (page 4) the single instruction *first* := *first.right* whose semantics is an example of case [S7], giving

$$\boxed{\overline{remove\_front} = [\,first := first\,.\,right]}$$

So we have to prove

$$\boxed{[\,first := first.right] \blacksquare\ model = model.tail}$$

that is to say, the property [T29] as proved in the preceding section.

Preservation of the acyclicity invariant [A30] follows from the property that if *f*** contains no cycle neither does its tail. Preservation of the no-tail-sharing invariant [A31] follows from the property that replacing a sequence by its tail cannot introduce tail sharing. (These properties can be made more formal and proved in the style of the properties in section 7.)

10 PROVING CORRECTNESS OF LIST REVERSAL

We now turn to a more sophisticated algorithm, list reversal (given on page 7). As originally noted, the result to be proved (apart from termination, and preservation of the class invariants) is that the body preserves the loop invariant, which read

$$\text{[I32]} \quad \textit{spliced}\colon \textbf{old}\ \textit{model} = \textit{first}\bullet\textit{right}^{**} \ \square\ \textit{mirror} \ + \ \textit{next}\bullet\textit{right}^{**}$$

We have to prove that

$$\overline{\textit{spliced}} \quad \overset{*}{\Rightarrow} \quad \overline{\textit{Body}} \blacksquare \overline{\textit{spliced}}$$

where *Body* is the body of the loop:

```
[previous, first, next] := [first, next, next.right]    -- Shift
first.put_right (previous)                              -- Reattach
```

Let us compute $\overline{\textit{Body}} \blacksquare \textit{spliced}$. From the instruction sequencing rule [S8] it is $[\overline{\textit{Shift}} \blacksquare \overline{\textit{Reattach}}] \blacksquare \textit{spliced}$ where *Shift* and *Reattach* are the two instructions.

Associativity applies so we first compute $\overline{\textit{Reattach}} \blacksquare \textit{spliced}$. From [S12], $\overline{\textit{Reattach}}$ is $\overline{\textit{first}} \blacksquare \overline{\textit{put\_right}}\ (\overline{\textit{previous}})$. Procedure *put_right* (*x*), in class *LINKABLE*, performs the assignment *right* := *x*, so its semantics $\overline{\textit{put\_right}}$ is, from the assignment rule [S6] and the procedure rule [S10]

$$\overline{\textit{put\_right}} \ = \ \textbf{function}\ a \mid \overline{\textit{right}} := \overline{a}$$

Combining this with the qualified call rule [S12] gives:

$$\overline{\textit{Reattach}} \ = \ \overline{\textit{first}} \blacksquare [\overline{\textit{right}} := \overline{\textit{previous}}]$$

Applying this to $\overline{\textit{spliced}}$ and retaining the **old** expression as per [S18] yields

$$\begin{aligned}
&\overline{\textit{Reattach}} \blacksquare \textit{spliced} \ = \\
&\qquad [\textbf{old}\ \textit{model} = [\textit{first} \blacksquare [\textit{right} := \textit{previous}]] \ \blacksquare \\
&\qquad\qquad [\textit{first}\bullet\textit{right}^{**} \ \square\ \textit{mirror} + \ \textit{next}\bullet\textit{right}^{**}]]
\end{aligned}$$

Distributing over + and applying [T25]:

$\overline{Reattach}$ ▪ *spliced* =
  [**old** *model* = [<*first*> + *previous* . *right***] ▫ *mirror* +
    next . *right***]

What we are computing is $\overline{Shift}$ ▪ $\overline{Reattach}$ ▪ *spliced*, so we must compose $\overline{Shift}$ with the right-hand side. From the multiple assignment axiom [S7]:

$\overline{Shift}$ = [*previous, first, next* := *first, next, next*.*right*]

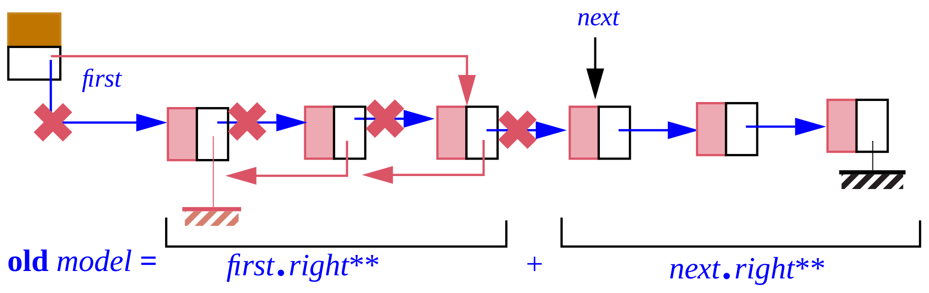
so that, applying the property [A4] of multiple assignment to all operands:

$\overline{Body}$ ▪ *spliced* =
  [**old** *model* = [<*next*> + *first* . *right***] ▫ *mirror* +
    next . *right* . *right***]

so that from the property [T26] of *mirror* we may write $\overline{Body}$ ▪ *spliced* as

[I33] **old** *model* = [*first* . *right***] ▫ *mirror* +
    <*next*> + *next* . *right* . *right***]

That this is an immediate consequence of the loop invariant *spliced* is clear from the picture that illustrated the invariant



List reversing*: *intermediate state

and is proved by using [T22] to simplify the second line of [I33], giving

$$\boxed{\begin{array}{ll}\text{[I34]} & \textbf{old}\ model = [first \bullet right^{**}] \ \square\ mirror\ + \\ & next \bullet right^{**}\end{array}}$$

which is the original invariant *spliced* [I32]. So we have proved that $\overline{spliced}$ implies $\overline{Body} \blacksquare \overline{spliced}$.

To prove termination we may use a similar technique to compute $\overline{Body} \blacksquare [next \bullet right^{**} \bullet count]$, the value of the loop variant (the length of the sequence $next \bullet right^{**}$) after an execution of the loop body, and find that it is one less than the initial value.

Preservation of the class invariants follows from the property that replacing the *model* by its reversed form, as expressed by the postcondition, cannot introduce any cycle or tail sharing.

The proof of procedure *put_front* or other routines that create object present no particular difficulty but needs the associated modeling of object creation and management discussed in [13].

11 CONCLUSION AND PLAN OF WORK

The approach described here appears to provide a workable basis for a systematic effort at proving the classes of a contracted library such as EiffelBase, covering the fundamental structures that application developers use daily.

The process

The example of class *LINKED_LIST* suggests a standard approach for proving library classes.

P1 • **Devise a model**. Choose a mathematical structure that will support expressing the properties of the instances of the class.

P2 • **Build a static theory**. In this step one must explore the properties of the model in a fixed state, independently of any execution (hence the term "static"), and prove them. We have seen typical examples of such properties: for sequences, [T26] stating that the mirror of a concatenation $s1 + s2$ is the concatenation of the mirror of $s2$ and the mirror of $s1$; the properties of sequence closure in section 7; and the properties of linked lists, such as acyclicity and non-tail sharing in section 8.

P3 • **Extend the contracts**. Typical contracts, written without the benefit of a model, only include a subset of the relevant properties. More precisely, preconditions must be exhaustive — otherwise the class is not safely usable — but postconditions and class invariants often miss important information that are hard to express without a model, for example, in an insertion operation, that all previous elements are still there. Loop invariants and variants are often omitted. All these must be filled in.

P4 • **Translate the class to mathematical form**. The denotational semantics of section **6** is the basis here. This step should be performed by an automated tool relying on a parser of the source language.

P5 • **Perform the proofs**. Although this paper has used a manual approach, the intent is to perform proofs mechanically; this explains the need for the previous step, since a proof tool will need to manipulate formulae expressed in an appropriate notation. The mechanically-checked proof effort may still, of course, require substantial manual support.

In line with the rest of the present discussion, this description covers proofs of individual classes. The framework described in [14], taking advantage of inheritance, involves both an effective (concrete) class such as *LINKED_LIST* and its deferred (abstract) ancestors, such as *LIST* describing general lists independent of an implementation. In this case there may be both an abstract model and a concrete one, requiring two extra steps:

P6 • **Prove that the abstract assertions imply the model assertions** in the deferred class.

P7 • **Prove the consistency of the concrete model against the abstract model** in the effective class.

Contrary to appearances this actually simplifies the process, since step P6, in the case of multiple descendants describing specific implementations of an abstract structure, moves up to the common ancestor part of the work that would have to be done anew for each descendant. See [14] for details.

Even for a single class, the process is unlikely to be strictly sequential. The proof step P5 may in particular encounter obstacles that require refining the model (step P1) or proving new properties of it (step P2).

One may also need to go back to the class texts. It is well known that the prospect of picking an ordinary piece of software and proving its correctness is an illusion: the software must have been written with correctness proofs in mind. We are starting from a better situation than usual since our target is the EiffelBase library, equipped with extensive contracts that are part of the design and documentation, not an afterthought, and indeed the idea of possible proofs has been there from the beginning. But we still expect that the proof process will require — aside from the correction of any actual bugs that it might uncover — simplifications and other changes to EiffelBase as it exists today.

Other object-oriented mechanisms

The present discussion has not accounted for classes, genericity, inheritance, the resulting type system, and dynamic binding. To add these mechanisms, the envisioned strategy is: introduce the notion of class, with room for generic parameterization, into the model; include support for expressing the inheritance relation between classes; and add a function *generator* that, for any object, gives the corresponding type (class plus actual generic parameters if any). The generator is set on object creation and does not change thereafter. One of the basic type rules is that for each attribute function f there is a type T such that

$$\boxed{\textbf{domain}\ (f) \subseteq \mathit{instances}\ (\bullet\ \{T\}\ \bullet)}$$

where *instances* is the inverse of *generator*. Note subset operator rather than equality, to account for possibly void references; for expanded attributes, which can't be void, it's an equality. The other significant change to the model of the present paper is that in the interpretation $\overline{x}\bullet\overline{f}\ (\overline{a})$ of a feature call $\overline{f}$ is obtained no longer directly from f but as $\mathit{dynamic}\ (f, \mathit{generator}\ (x))$ where the function *dynamic*, accounting for dynamic binding, yields the version of a certain feature for a certain type.

Future work

Aside from the extension of the model to cover the whole of object-oriented programming, the tasks lying ahead are clear: apply the above process to a growing set of classes covering the fundamental data structures and algorithms of computing science. This involves building models, developing the associated theories, completing the contracts of the corresponding classes, attempting the proofs, and refining the library in the process.

Acknowledgments

An early presentation of the results reported here benefited from important comments by Robert Stärk and Armin Biere, as well as Bernd Schoeller who also provided useful feedback on an early draft of this article. Emil Sekerinski's criticism led to the correction of a significant error. The article benefited from a detailed reading and important suggestions (not all yet taken into account at this stage of the work) from Peter Müller, who also pointed out some existing work that should be referenced, and Robert Switzer. Finally, a presentation at a meeting of the IFIP Working Group on Programming methodology (WG2.3) was the occasion of invaluable criticism from the group's members.

REFERENCES

[1] Martín Abadi and Luca Cardelli: *A Theory of Objects*, Monographs in Computer Science, Springer-Verlag, 1996.

[2] Jean-Raymond Abrial, *The B Book*, Cambridge University Press, 1995.

[3] Ralph Back, X. Fan and Viorel Preoteasa: *Reasoning about Pointers in Refinement Calculus*, Technical Report, Turku Centre for Computer Science, Turku (Finland), 22 August 2002.

[4] Richard Bornat: *Proving Pointer Programs in Hoare Logic*, in *Mathematics of Program Construction*, Springer-Verlag, 2000, pages 102-106.

[5] ClearSy [name of company, no author listed]: Web documents on Atelier B, www.atelierb.societe.com, last consulted December 2002.

[6] C.A.R. Hoare: *Proof of Correctness of Data Representations*, in *Acta Informatica* 1 (1972), pp. 271--281. Also in C.A.R. Hoare and C. B. Jones (ed.): *Essays in Computing Science*, Prentice Hall International, Hemel Hempstead (U.K.), 1989, pages 103-115.

[7] C. A. R. Hoare. *Procedures and parameters*: *An axiomatic approach*. In E. Engeler, editor, *Symposium on Semantics of Algorithmic Languages*, volume 188 of Lecture Notes in Mathematics, pages 102-116. Springer-Verlag, 1971.

[8] C.A.R. Hoare and He Jifeng: *A Trace Model for Pointers*, in *ECOOP '99 — Object-Oriented Programming*, Proceedings of 13th European Conference on Object-Oriented Programming, Lisbon, June 1999, ed. Rachid Guerraoui, Lecture Notes in Computer Science 1628, Springer-Verlag, pages 1-17.

[9] Bertrand Meyer: *Introduction to the Theory of Programming Languages*, Prentice Hall, 1990.

[10] Bertrand Meyer: *Object-Oriented Software Construction, 2nd edition*, Prentice Hall, 1997.

[11] Bertrand Meyer, Christine Mingins and Heinz Schmidt: P*roviding Trusted Components to the Industry*, in *Computer* (IEEE), vol. 31, no. 5, May 1998, pages 104-105.

[12] Bertrand Meyer et al.: Trusted Components papers at se.inf.ethz.ch, last consulted December 2002.

[13] Bertrand Meyer: *Proving Pointer Program Properties*, series of columns to appear in *Journal of Object Technology*, draft version available at www.inf.ethz.ch/~meyer/ongoing/references/, last consulted January 2003.

[14] Bertrand Meyer: *A Framework for Proving Contract-Equipped Classes*, to appear in *Abstract State Machines 2003 - Advances in Theory and Applications*, Proc. 10th International Workshop, Taormina, Italy, March 3-7, 2003, eds. Egon Boerger, Angelo Gargantini, Elvinia Riccobene, Springer-Verlag 2003. Pre-publication copy at www.inf.ethz.ch/~meyer/publications/, last consulted January 2003.

[15] Bernhard Möller: *Calculating with Pointer Structures*, in *Algorithmic Languages and Calculi*, Proceedings of IFIP TC2/WG2.1 Working Conference, Le Bischenberg (France), February 1997, Chapman and Hall, 1997, pages 24-48.

[16] Joseph M. Morris, *A general axiom of assignment*; *Assignment and linked data structures*; *A proof of the Schorr-Waite algorithm*. In *Theoretical Foundations of Programming Methodology*, Proceedings of the 1981 Marktoberdorf Summer School, eds. Manfred Broy and Gunther Schmidt, Reidel 1982, pages 25-51.

[17] John C. Reynolds: *Separation Logic*: *A Logic for Shared mutable Data Structures*, in Proceedings of 17th Annual IEEE Symposium on Logic in Computer Science, Copenhagen, July 22-25 2002.

[18] Norihisha Suzuki, *Analysis of Pointer "Rotation"*, in *Communications of the ACM*, vol. 25, no. 5, May 1982, pages 330-335.