A semantics for proving class correctness
Draft

Pablo Castro
Departamento de Computación
Universidad Nacional de Río Cuarto
Argentina
e-mail: pcastro@dc.exa.unrc.edu.ar

Javier Blanco
Departamento de Computación
Universidad Nacional de Córdoba- FaMAF
Argentina
e-mail: blanco@mate.uncor.edu

release 20-08-2005

Contents

1 Introduction

In [2], Bertrand Meyer takes the initial steps towards a theory for proving contract equipped class correctness (see also [3] and [4]). The aim of this framework is to develop trusted components, i.e. a set of classes which have been proved correct with respect to their contracts in order to incrementally build safer software. Meyer formalizes object-oriented concepts such as references, classes, objects and method calls trying to keep the theory as simple as possible, using only set theory and higher order (partial) functions. However, the formalism is only sketched and some concepts have to be refined in order to be applied to concrete examples, in particular the semantics of routine calls.

In this work we review the ideas from [2]. Our main goal is to completely define the semantics to make it workable for proving concrete examples. Furthermore, we compare this approach with other works on pointers [5] showing that it is possible to rephrase Bornat's axioms in Meyer's framework. Finally, we present some applications of the semantics for the verification of an Eiffel's library.

2 Notation

In this article we will use the following symbols mostly taken from Meyer and inspired in the B or Z notation.

$\triangleq$	Equality by definition.
$\cup, \cap, \times$	Union, intersection and cartesian product.
Π_n	Projection.
$A \to B$	Function space with domain A and codomain B.
$A \nrightarrow B$	Partial function space with domain A y codomain B.
$+$	Disjoint set union.
$\mathbb{P}, \mathbb{F}$	Powerset and set of finite subsets.
$\mathbb{N}$	Natural numbers set.
$=$	Strong equality, i.e., given two (possibly undefined) expressions e_1, e_2, we say that $e_1 = e_2$ holds iff both expressions are defined and them denote the same value or both are undefined.

Following Meyer's conventions, we indicate the arguments of a function between brackets and use square brackets to group expressions. Furthermore, we use λ-expressions for defining functions but with the word function instead of λ

3 Basic definitions

In this section we formally describe the object oriented terms, mainly taken from [2]

3.1 Function composition

We formalize references by means of functions. In general a reference can be described by a function with type $Objects \twoheadrightarrow States \twoheadrightarrow Objects$ where $Objects$ is the set of objects in the system and $States$ the set of its possible states. A state is composed by a mapping from names into functions from objects to objects and a set of objects indexed by natural numbers which denotes the set of active objects in the state. On the other hand, those instructions which modify the state of the system are modeled as functions of type $Objects \twoheadrightarrow States \twoheadrightarrow States$, i.e. functions which given an object and the current state produce a new state after the execution of the instruction on this object. Functional composition has a very important role in the description of a complex system, for example a sequential composition is naturally described as a composition of state transformers. Since object oriented computations has the overhead that all states are indexed by objects the compositions are less straightforward and must be dealt with carefully. We give definitions for different kinds of compositions following [2]

Definition 1 ($\Box$) *Given functions $f : A_1 \twoheadrightarrow A_2 \twoheadrightarrow ... \twoheadrightarrow A_n \twoheadrightarrow B$ and $g : B \twoheadrightarrow C$, we define their composition $\Box$ as follows:*

$$f \,\Box\, g \triangleq \mathsf{function}\ a_1 \mid ... \mid \mathsf{function}\ a_n \mid g([[[f(a_1)...](a_n)]])$$

In other words $\Box$ is a generalization of the usual composition (in diagrammatic order). This not only allows for application to many arguments but also n can be 0 in which case this composition reduces to application of a function to a constant. For example, if $n \in Nat$ y $f : Nat \to Nat$, then $f(n) = n \,\Box\, f$

Definition 2 ($\bullet$) *Given functions$f : A \twoheadrightarrow B \twoheadrightarrow C$ and $g : C \twoheadrightarrow B \twoheadrightarrow D$, their composition $\bullet$ is defined as:*

$$f \bullet g \triangleq \mathsf{function}\ x \mid \mathsf{function}\ s \mid g(f(x)(s))(s)$$

The operator $\bullet$ allows to compose two functions with a common second argument.

Definition 3 ($\blacksquare$) *Given functions $f : A_1 \twoheadrightarrow A_2 \twoheadrightarrow ... \twoheadrightarrow A_n \twoheadrightarrow A \twoheadrightarrow B$ and $g : A_1 \twoheadrightarrow A_2 \twoheadrightarrow ... \twoheadrightarrow A_n \twoheadrightarrow B \twoheadrightarrow C$, we define their composition $\blacksquare$ as follows:*

$$f \blacksquare g \triangleq \mathsf{function}\ a_1 \mid ... \mid \mathsf{function}\ a_n \mid [[[f(a_1)](a_2)]...(a_n)] \,\Box\, [[[g(a_1)](a_2)]...(a_n)]$$

Intuitively, operation $\blacksquare$ compose functions which have n initial arguments in common.

3.2 References, objects, states and attributes

In order to define the semantics of an object-oriented program we need to introduce the following sets

- *Address*, set of all possible memory addresses.
- *Objects*, (Finite) set of all addresses which may contain an object, i.e.: $Objects \subseteq Address$.

- *Expanded*, is the set of values which are not references, including integer, boolean, reals, etc.
- $Values$, denotes the set of all possible values, i.e.: $Values \triangleq Objects \cup Expanded$.
- $Names$, is the set of syntactical names of attributes and variables in an Eiffel code.

As mentioned before, the set of all possible states is called *States*. Each state is defined through a collection of objects and a collection of functions each of which can be a reference to another object or an expanded value. Formally, this is defined as follows:

$$Ref \triangleq Objects \nrightarrow Objects \tag{1}$$

On the other hand, the attributes that are not references are defined as

$$NonRef \triangleq Objects \nrightarrow Expanded \tag{2}$$

Hence, the state space is the following set:

$$States \triangleq [Names \nrightarrow [Ref + NonRef]] \times [Nat \nrightarrow Objects] \tag{3}$$

Intuitively, a state of a purely object oriented program is given by the set of created objects and a set of functions corresponding to the attributes of each object.

In order to have let our formalism to be scalable, we need to be able to reason globally about references without referring to any particular state. This is achieved by a similar idea to the *component- as-array* trick as described in [5]. The references can be extended to functions of the type $Objects \nrightarrow States \nrightarrow Objects$ or $Objects \nrightarrow States \nrightarrow Expanded$, defining for any $f \in Names$, the following function:

$$\overline{f} \triangleq \mathsf{function}\ o \mid \mathsf{function}\ s \mid \Pi_1(s)(f)(o) \tag{4}$$

Depending on f the type of $\overline{f}$ will be $Objects \nrightarrow States \nrightarrow Objects$ for references and $Objects \nrightarrow States \nrightarrow Expanded$ for non-references.

An interesting property of $\bar{}$ is the following:

Property 1 ($\mathbf{P\bar{}1}$) *For every $s \in States$, $o \in Objects$, $f \in Names$ it holds that:*

$$[\Pi_1(s)](f)(o) = \overline{f}(o)(s)$$

Proof.

Immediate by definition.

Q.E.D.

We can consider a function **void** : $Objects \nrightarrow States \nrightarrow Objects$, which is undefined for every element in *Objects*. This function is useful to define the semantics of *void*; Meyer does not consider a value *void* assuming that it is modeled by leaving the function undefined. However, sometimes having an explicit *void* value is desirable, for example when a test for void is used in

a condition or when the value *void* is assigned to an attribute. The definition of **void** is given through the following property:

$$[[f := \textbf{void}] \blacksquare f] \; is \; undefined \; for \; every \; o \in Objects \tag{5}$$

We use the **void** in boldface to denote the semantic value and keep the *void* for the syntactic reserved word.

3.3 Expanded elements as references

In this section we define the pointwise extension of the boolean operations to functions of type $Objects \nrightarrow States \nrightarrow Bool$.

- **Implication:** If $p : Objects \nrightarrow States \nrightarrow Bool$ y $q : Objects \nrightarrow States \nrightarrow Bool$ then:

  $p \overset{*}{\Rightarrow} q \overset{\triangle}{=} \mathsf{function}\ o \mid \mathsf{function}\ s \mid [p(o)(s)] \Rightarrow [q(o)(s)]$

- **Conjunction:** If $p : Objects \nrightarrow States \nrightarrow Bool$ y $q : Objects \nrightarrow States \nrightarrow Bool$ then:

  $p \overset{*}{\wedge} q \overset{\triangle}{=} \mathsf{function}\ o \mid \mathsf{function}\ s \mid [p(o)(s)] \wedge [q(o)(s)]$

- **Negation:** If $q : Objects \nrightarrow States \nrightarrow Bool$ then:

  $\overset{*}{\neg} q \overset{\triangle}{=} \mathsf{function}\ o \mid \mathsf{function}\ s \mid \neg[q(o)(s)]$

- **True:**

  $\overset{*}{true} \overset{\triangle}{=} \mathsf{function}\ o \mid \mathsf{function}\ s \mid true$

- **Equality:** It is the classical pointwise extension of **strong** equality:

  $[f \overset{*}{=} g] \overset{\triangle}{=} \mathsf{function}\ o \mid \mathsf{function}\ s \mid f(o)(s) = g(o)(s)$

- **Quantifiers:** Let $e : Objects \nrightarrow States \nrightarrow Bool$ be an expression with a free variable x of type A, and Q a quantifier ($\exists$ or $\forall$) then:

  $\overset{*}{Q}\, x : A \bullet e \overset{\triangle}{=} \mathsf{function}\ o \mid \mathsf{function}\ s \mid Qx : A \bullet [e(o)(s)]$

These operations extend the classical boolean operations in a similar way as the boolean structures from [**?**]. Note that conjunction, implication and negation are strict operators (i.e., they are undefined when any of their parameters is undefined).

We call $\overset{*}{Bool}$ to the pointwise extension of the boolean type ($Object \nrightarrow States \nrightarrow Bool$). We say that a function $\phi \in \overset{*}{Bool}$ is **valid** if and only if $\forall o : Dom(\phi), \forall s : Dom(\phi(o)) : \phi(o)(s)$.

Note that from formulae in $\overset{*}{Bool}$ and functions of type $Objects \nrightarrow States \nrightarrow States$ one can construct new functions of type $Objects \nrightarrow States \nrightarrow Bool$ by using $\blacksquare$. In the same way we consider the pointwise extension of types like Nat or Seq. Some examples are:

$\overset{*}{5}$, is the pointwise extension of number 5.

$\overset{*}{\langle\rangle}$, is the pointwise extension of empty sequence.

$\overset{*}{+}$, is the pointwise extension of the concatenation operator on sequences.

Note that if c is a constant of *Expanded* then $\overset{*}{c}$ is a constant function of type $Objects \nrightarrow States \nrightarrow Expanded$. Summarizing, in our formalism expressions like $5 + (9 - 10)$ are written as follows: $\overset{*}{5}\overset{*}{+}\ (\overset{*}{9}\overset{*}{-}\overset{*}{10})$. An important property of the pointwise extensions is the following:

Theorem 1 (Dist.■) *Let ϕ be an operator of signature $A_1 \times ... \times A_n \to A$, and $h : Objects \nrightarrow States \nrightarrow States$, then:*

$$h \blacksquare \overset{*}{\phi}\ (\overset{*}{a_1}, ..., \overset{*}{a_n}) = \overset{*}{\phi}\ (h \blacksquare \overset{*}{a_1}, ..., h \blacksquare \overset{*}{a_n})$$

Where: $\overset{*}{a_1}$: $Objects \nrightarrow States \nrightarrow A_1$,..., $\overset{*}{a_n}$: $Objects \nrightarrow States \nrightarrow A_n$.

Proof.

We apply the left expression to an arbitrary object and an arbitrary state.

$[h \blacksquare \overset{*}{\phi}\ (\overset{*}{a_1}, ..., \overset{*}{a_n})](o)(s)$
$=$ { *Definition of* ■ }
$[\overset{*}{\phi}\ (\overset{*}{a_1}, ..., \overset{*}{a_n})](o)(h(o)(s))$
$=$ { *Definition of* $\overset{*}{\phi}$ }
$\phi([\overset{*}{a_1}\ (o)(h(o)(s))], ..., [\overset{*}{a_n}\ (o)(h(o)(s))])$
$=$ { *Definition of* ■ }
$\phi([h \blacksquare \overset{*}{a_1}](o)(s), ..., [h \blacksquare \overset{*}{a_n}](o)(s))$
$=$ { *Definition of* $\overset{*}{\phi}$ }
$\overset{*}{\phi}\ ([h \blacksquare \overset{*}{a_1}], ..., [h \blacksquare \overset{*}{a_n}])(o)(s)$

Q.E.D.

Graphically we can say that the diagram of figure 1 conmutes.

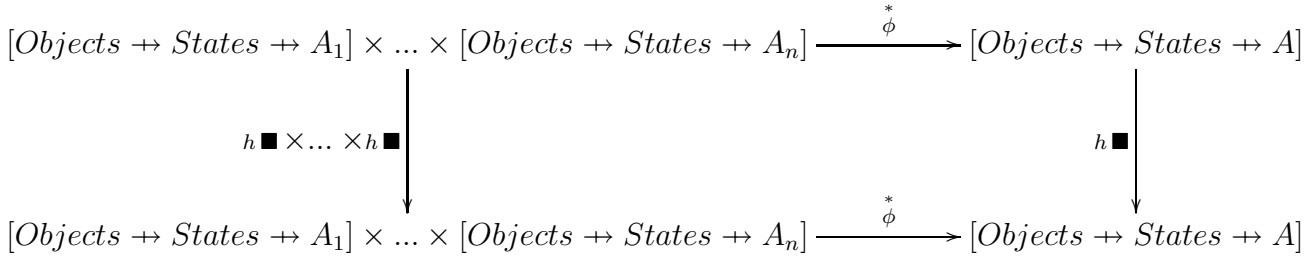

Figure 1: Diagram of theorem 1

In order to simplify our notation we avoid the explicit indication of pointwise extension when possible.

3.4 Assignment

The most important sentence that actually changes the state is the assignment statement. Hence, we need an operation in our semantic domain which performs this state change. With this goal in mind, Meyer defines the following function:

$$[f := g] : Objects \nrightarrow States \nrightarrow States \tag{6}$$

Intuitively, this function takes an object obj and a state s, and returns a new state s' in which only f changes and only for the given object obj in the following way:

$$f(obj)(s') = g(obj)(s) \tag{7}$$

In [2] there is no formal definition of this function and hence the proof of its properties is not rigorous. Since we are trying to have a complete formal definition of the language we introduce an update operator to formally define the assignment:

$$f \oplus_s o \mapsto E \tag{8}$$

This operator modifies the function f in the state s only for the object o in which case it returns the value associated to the function E, which can be a reference or an expanded value. The overloading of function $\bar{}$ is useful here to avoid case analysis

Definition 4 ($\oplus$) *Let $s \in States$ and $f \in Names$ with $\Pi_1(s)(f) : Objects \nrightarrow B$, then a function $[f \oplus_s o \mapsto E] : Names \nrightarrow [Ref + NonRef]$, with $o \in Objects$ and $E : Objects \nrightarrow States \nrightarrow B$, is defined as follows:*

$$[f \oplus_s o \mapsto E](h)(o') \stackrel{\wedge}{=} \mathsf{if}\ [o = o'] \wedge [f = h]\ \mathsf{then}\ E(o)(s)\ \mathsf{else}\ [\Pi_1(s)](h)(o')$$

Some properties of $\oplus$ are:

Property 2 ($P \oplus 1$)

$$[f \oplus_s o \mapsto E](f)(o) = E(o)(s)$$

Proof.
Immediate by definition of $\oplus$.
Q.E.D.

Property 3 ($P \oplus 2$) *For every $f, g \in Names$, with $f \neq g$ or $o \neq o'$, it holds that:*

$$[f \oplus_s o \mapsto E](g)(o') = [\Pi_1(s)](g)(o')$$

Proof.
Immediate by definition of $\oplus$.
Q.E.D.

Now we can use $\oplus$ in order to define the assignment $(:=)$ for every $f \in Names$ and $g : Objects \nrightarrow States \nrightarrow Values$:

$$[\overline{f} := g] \stackrel{\wedge}{=} \mathsf{function}\ o\ |[\ \mathsf{function}\ s \mid \langle [f \oplus_s o \mapsto g], \Pi_2(s) \rangle \tag{9}$$

Some important properties that $:=$ inherits from $\oplus$ are the following.

Property 4 (P:=1) *For every $f \in Names$:*

$$\overline{f}(o)([\overline{f} := g](o)(s)) = g(o)(s)$$

Proof.
$\overline{f}(o)([\overline{f} := g](o)(s))$
$=$ { *def.* $(:=)$ }
$\overline{f}(o)(\langle [f \oplus_s o \mapsto g], \Pi_2(s) \rangle)$
$=$ { *def.* $\overline{f}$ }
$[\Pi_1(\langle [f \oplus_s o \mapsto g], \Pi_2(s) \rangle)](f)(o)$
$=$ { *def.* Π_1 }
$[f \oplus_s o \mapsto g](f)(o)$
$=$ { *P⊕1* }
$g(o)(s)$
Q.E.D.

Property 5 (P:=2) *For every $f, h \in Names$ and function g, with $f \neq h$ or $o' \neq o$, where $o, o' \in Objects$:*

$$\overline{h}(o')([\overline{f} := g](o)(s)) = \overline{h}(o')(s)$$

Proof.
$\overline{h}(o')([\overline{f} := g](o)(s))$
$=$ { *def.* $(:=)$ }
$\overline{h}(o')(\langle [f \oplus_s o \mapsto g], \Pi_2(s) \rangle)$
$=$ { *def.* $\overline{h}$ }
$[\Pi_1(\langle [f \oplus_s o \mapsto g], \Pi_2(s) \rangle)(h)(o')$
$=$ { *def.* Π_1 }
$[f \oplus_s o \mapsto g](h)(o')$
$=$ { *P⊕2* }
$[\Pi_1(s)](h)(o')$
$=$ { *def.* }
$\overline{h}(o')(s)$
Q.E.D.

We can now prove rigorously properties $A1$ and $A2$ from [2]

Property 6 (A1) *For all $f \in Names$ and $g : Objects \nrightarrow States \nrightarrow Objects$:*

$$[\overline{f} := g] \blacksquare \overline{f} \stackrel{*}{=} g$$

Proof.
We apply the left hand side to an arbitrary object o, and we get by definition of $\blacksquare$:

$[\overline{f} := \overline{g}](o) \,\square\, \overline{f}(o)$
$=$ { *def.*$\square$ }
function $s \mid \overline{f}(o)([\overline{f} := g](o)(s))$
$=$ { *P:=1* }
function $s \mid g(o)(s)$
$=$ { *extensionality* }
$g(o)$
Q.E.D.

Property 7 (A2) *For any* $f, h \in Names$ *with* $f \neq h$ *it holds that:*

$$[\overline{f} := g] \blacksquare \overline{h} \stackrel{*}{=} \overline{h}$$

Proof.
We apply the left hand side to an arbitrary o, and get by definition of $\blacksquare$:

$[\overline{f} := \overline{g}](o) \,\square\, \overline{h}(o)$
$=$ { *def.*$\square$ }
function $s \mid \overline{h}(o)([\overline{f} := g](o)(s))$
$=$ { *P:=2* }
function $s \mid \overline{h}(o)(s)$
$=$ { *extensionality* }
$h(o)$
Q.E.D.

In the next theorems and properties we will use the operator $\bar{}$ only to avoid ambiguities. Some other properties of $:=$ and $\blacksquare$ which are useful are.

Property 8 (P $\blacksquare$ 1) *: For all* $h, f \in Names$ *with* $h \neq f$*:*

$$[\overline{f} := g] \blacksquare [\overline{f} \bullet \overline{h}] \stackrel{*}{=} g \bullet \overline{h}$$

Proof.
$[f := g] \blacksquare [f \bullet h]$
$=$ { *def.* $\blacksquare$ }
function $o \mid [f := g](o) \,\square [f \bullet h](o)$
$=$ { *def.* $\bullet$ *and def.* $\square$ }
function $o \mid$ function $s \mid [h(f(o)([f := g](o)(s)))]([f := g](o)(s))$
$=$ { *def.* $\square$ }
function $o \mid$ function $s \mid [h([[f := g](o) \,\square f(o)])(s)]([f := g](o)(s))$
$=$ { *def.* $\blacksquare$ }
function $o \mid$ function $s \mid h([[f := g] \blacksquare f](o)(s))([f := g](o)(s))$

$=$ { *Prop. A1* }
function o | function s | $h(g(o)(s))([f := g](o)(s))$
$=$ { $P := 2$ }
function o | function s | $h(g(o)(s))(s)$
$=$ { *def.* $\bullet$ }
$g \bullet h$
Q.E.D.

Property 9 (P■ 2) *: For all $f, g \in Names$ with $f \neq g$:*

$$[\overline{f} \bullet [\overline{g} := h]] \blacksquare [\overline{f} \bullet \overline{g}] \stackrel{*}{=} \overline{f} \bullet h$$

Proof.

We apply the first expression to an arbitrary object and an arbitrary state:

$[f \bullet g](o)([f \bullet [g := h]](o)(s))$
$=$ { *def.* $\bullet$ }
$[f \bullet g](o)([g := h](f(o)(s))(s))$
$=$ { *def.* $\bullet$ }
$g(f(o)([g := h](f(o)(s))(s)))([g := h](f(o)(s))(s))$
$=$ { $f \neq g$ *and* $P := 2$ }
$g(f(o)(s))([g := h](f(o)(s))(s))$
$=$ { *def.* ■ }
$[[g := h] \blacksquare g](f(o)(s))(s)$
$=$ { *Prop. A1* }
$h(f(o)(s))(s)$
$=$ { *def.* $\bullet$ }
$[f \bullet h](o)(s)$
Q.E.D.

Property 10 (P■ 3) *: For all $t, g, f \in Names$ with $t \neq g$ and $g \neq f$ it holds that:*

$$[\overline{f} \bullet [\overline{g} := h]] \blacksquare [\overline{f} \bullet \overline{t}] \stackrel{*}{=} [\overline{f} \bullet \overline{t}]$$

Proof.

$[[f \bullet [g := h]] \blacksquare [f \bullet t]](o)(s)$
$=$ { *def.* ■ }
$[f \bullet t](o)([f \bullet [g := h]](o)(s))$
$=$ { *def.* $\bullet$ }
$[f \bullet t](o)([g := h](f(o)(s))(s))$
$=$ { *def.* $\bullet$ }
$t(f(o)([g := h](f(o)(s))(s)))([g := h](f(o)(s))(s))$
$=$ { *Prop. P := 2 and Hip.* }
$t(f(o)(s))(s)$

$=$ $\{$ *def.* $\bullet$ $\}$
$[f \bullet t](o)(s)$
Q.E.D.

The following property is important since it allows us to reason about the "dot" operator.

Property 11 (P■4) *For all functions* $p : Objects \nrightarrow States \nrightarrow Objects$, $f : Objects \nrightarrow States \nrightarrow States$ *and* $g : Objects \nrightarrow States \nrightarrow B$ *it holds that:*

$$[[p \bullet f] \blacksquare p \overset{*}{=} p] \overset{*}{\Rightarrow} [p \bullet [f \blacksquare g] \overset{*}{=} [p \bullet f] \blacksquare [p \bullet g]]$$

Proof.

$[[p \bullet f] \blacksquare [p \bullet g]](o)(s)$
$=$ $\{$ *def.*■ $\}$
$[p \bullet g](o)([p \bullet f](o)(s))$
$=$ $\{$ *def.* $\bullet$ $\}$
$[p \bullet g](o)(f(p(o)(s))(s))$
$=$ $\{$ *def.* $\bullet$ $\}$
$g(p(o)(f(p(o)(s))(s)))(f(p(o)(s))(s))$
$=$ $\{$ *def.*■ $\}$
$g([p \bullet f \blacksquare p](o)(s))(f(p(o)(s))(s))$
$=$ $\{$ *hyp.* $\}$
$g(p(o)(s))(f(p(o)(s))(s))$
$=$ $\{$ *def.*■ $\}$
$[f \blacksquare g](p(o)(s))(s)$
$=$ $\{$ *def.* $\bullet$ $\}$
$[p \bullet [f \blacksquare g]](o)(s)$
Q.E.D.

Finally, the next properties show that elements in $Names$ satisfy the independence requirements established in [10].

Property 12 *For every* $f \in Names$ *it holds that:*

$$[[\overline{f} := x] \blacksquare y \overset{*}{=} y] \overset{*}{\Rightarrow} [[\overline{f} := x] \blacksquare [\overline{f} := y] \overset{*}{=} [\overline{f} := y]]$$

Proof.
Left as exercise.
Q.E.D.

Property 13 *For all* $f, g \in Names$ *with* $f \neq g$*:*

$$[[[\overline{f} := x] \blacksquare y \overset{*}{=} y] \overset{*}{\wedge} [[\overline{g} := y] \blacksquare x \overset{*}{=} x]] \overset{*}{\Rightarrow} [[\overline{f} := x] \blacksquare [\overline{g} := y] \overset{*}{=} [\overline{g} := y] \blacksquare [\overline{f} := x]]$$

Proof.
Left as exercise.
Q.E.D.

3.5 Freezing functions

Sometimes it will be useful to "freeze" a function in a given state and object. This operator will be important to define the semantics of procedures. We use $\widehat{\ }$ to note the "freezer" operator. Intuitively, given a function $f : Objects \nrightarrow States \nrightarrow Objects$, an object o and state s, $\widehat{f(o)(s)}$ is a constant function which evaluates to $f(o)(s)$ in every object and state. Its formal definition is:

$$\widehat{\ }(f)(o)(s) \triangleq \mathsf{function}\ o' \mid \mathsf{function}\ s' \mid f(o)(s) \tag{10}$$

Where: $f : Objects \nrightarrow States \nrightarrow A$, $o \in Objects$ and $s \in States$. For simplicity we introduce the following notation:

$$\widehat{f(o)(s)} \triangleq \widehat{\ }(f)(o)(s)$$

Some interesting properties on $\widehat{\ }$ are:

Property 14 *Let $x : Objects \nrightarrow States \nrightarrow A$, and $y : Objects \nrightarrow States \nrightarrow Objects$ then:*

1. $[\widehat{x(o)(s)}(o')(s')] \overset{*}{=} \widehat{x(o)(s)}$ *(recall that $\widehat{x(o)(s)} : Objects \nrightarrow States \nrightarrow A$)*
2. $y \bullet \widehat{x(o)(s)} \overset{*}{=} \widehat{x(o)(s)}$
3. *if $x(o)(s)$ is undefined then* $\widehat{x(o)(s)} \overset{*}{=}$ **void**
4. $\overset{*}{\widehat{c}}\,(o)(s) \overset{*}{=} \overset{*}{c}$ *(where $c \in Expanded$)*

for every $o \in Objects$ and $s \in States$.
Proof.

1)
We apply the right hand side to an arbitrary object and state:

$[\widehat{x(o)(s)}(o')(s')](o'')(s'')$
$=$ { *def.* $\widehat{\ }$ }
$\widehat{x(o)(s)}(o')(s')$
$=$ { *def.*$\widehat{\ }$ }
$x(o)(s)$
$=$ { *def.*$\widehat{\ }$ }
$\widehat{x(o)(s)}(o'')(s'')$

2)
We apply the right hand side to an arbitrary object and state:

$[y \bullet \widehat{x(o)(s)}](o')(s')$
$=$ { *def.* $\bullet$ }

$\widehat{x(o)(s)}(y(o')(s'))(s')$
$=$ { *def.*$\widehat{\ }$ }
$x(o)(s)$
$=$ { *def.*$\widehat{\ }$ }
$\widehat{x(o)(s)}(o')(s')$

3)
Straightforward by definition of strong equality.
4)
We apply the right hand side to an arbitrary object and state:

$\overset{*}{\widehat{c}}\,(o)(s)(o')(s')$
$=$ { *def.* $\widehat{\ }$ }
$\overset{*}{c}\,(o)(c)$
$=$ { *def.* * }
$\overset{*}{c}\,(o')(s')$
Q.E.D.

3.6 Assertions as state transformers

Most modern languages allow to incorporate assertions as part of the code. For example, the following is and extract of an Eiffel program:

```
...x:=y+1; check x>y end; y:=y+1...
```

The aim of this construction is to ensure that condition `y<x` holds after the execution of the assignment `x:=y+1`, otherwise an exception is raised. Whenever a component is formally verified, assertion statements can be deleted without any effect on the execution.

In our formalism, assertions are functions on the space $Objects \nrightarrow States \nrightarrow States$. More precisely, given $a \in \overset{*}{Bool}$ we define assertion $\{a\}$ in the following way:

$$\{a\} \triangleq \mathsf{function}\ o \mid \mathsf{function}\ s \mid \quad \mathsf{if}\ a(o)(s)\ \mathsf{then}\ s$$

Note that $\{a\}$ is undefined in the states where a does not hold. This can be understood as an assertion violation which will, in Eiffel, raise an exception.

Two main issues make assertions a key point of our *framework*:

- Assertions are useful to modularize proofs.
- Assertions simplify certain semantic aspects .

We will discuss these points in further sections. Now we give some properties that assertions enjoy:

Property 15 . $[h \blacksquare p] \overset{*}{\Leftrightarrow} [h \blacksquare \{p\} \overset{*}{=} h]$
Proof.
$\overset{*}{\Rightarrow}$)
Let o, s be any object and state. It holds that:

$$[h \blacksquare p](o)(s) = p(o)[h(o)(s)]$$

We unfold the left hand side
$[h \blacksquare \{p\}](o)(s)$
$=$ { *def.*$\blacksquare$ }
$\{p\}(o)[h(o)(s)]$
$=$ { *def.*$\{p\}$ *and hyp.* }
$h(o)(s)$

$\overset{*}{\Leftarrow}$)
Let o and s be any object and state respectively. If $h(o)(s)$ is undefined there is nothing to prove. Let $h(o)(s)$ be defined. Then, by hypothesis $h \blacksquare \{p\}$ is also defined, and hence it holds that $p(o)(h(o)(s))$ i.e $[h \blacksquare p](o)(s)$.
Q.E.D.

Note that the antecedent p is a function in $Objects \nrightarrow States \nrightarrow Bool$. However, in the consequent $\{p\}$ is a function in $Objects \nrightarrow States \nrightarrow States$. This can be interpreted as the fact that a correct assertion can be erased from the code.

Property 16 $h \blacksquare \{\overset{*}{true}\} \overset{*}{=} h$

Proof.
Immediate by definition of assertions.
Q.E.D.

In other words, $\overset{*}{true}$ is always true. Analogously, it can be shown that $\overset{*}{false}$ never holds.

Property 17 $h \blacksquare \{\overset{*}{false}\}$ *is undefined for any state and object.*
Proof.
Immediate by definition of assertions.
Q.E.D.

Theorem 2 (split) . $[[h \blacksquare p] \overset{*}{\wedge} [p \overset{*}{\Rightarrow} g]] \overset{*}{\Rightarrow} [h \blacksquare \{p\} \blacksquare g]$
Proof.
Let o, s be any object and state respectively. By hypothesis it holds that :

$p(o)(h(o)(s))$
$\Rightarrow$ { *Hip. and Def.*$\overset{*}{\Rightarrow}$ }

$g(o)(h(o)(s))$
$=$ { *def.*$\blacksquare$ }
$[h \blacksquare g](o)(s)$
$=$ { *property 15 and hyp.* }
$[h \blacksquare \{p\} \blacksquare g](o)(s)$

Q.E.D.

Property 18 $[p \stackrel{*}{\Rightarrow} q] \stackrel{*}{\Leftrightarrow} [\{p\} \blacksquare q]$
Proof.

Q.E.D.

Property 19 (weaken post) $[x \stackrel{*}{\Rightarrow} y] \stackrel{*}{\Rightarrow} [[p \blacksquare x] \stackrel{*}{\Rightarrow} [p \blacksquare y]]$

Proof.

Q.E.D.

Property 20 (strengthen pre) $[x \stackrel{*}{\Rightarrow} y] \stackrel{*}{\Rightarrow} [[\{y\} \blacksquare q] \stackrel{*}{\Rightarrow} [\{x\} \blacksquare q]]$

Proof.
If $[x(o)(s) = false]$ *then* $\{x\}(o)(s)$ *is undefined (by definition), and hence it is not relevant for the validity of the consequent.*
If $x(o)(s) = true$ *then by hypothesis also* $y(o)(s) = true$*, hence:*

$[\{x\} \blacksquare q](o)(s)$
$=$ { *def.* $\blacksquare$ }
$q(o)(\{x\}(o)(s))$
$=$ { $x(o)(s) = true$ *and Def.*$\{-\}$ }
$q(o)(s)$
$=$ { $y(o)(s) = true$ }
$q(o)(\{y\}(o)(s))$
$=$ { *Hyp.* }
$[\{y\} \blacksquare q](o)(s)$
$=$ { *hyp.* }
$true$
Q.E.D.

Property 21

- $x \bullet \{h \stackrel{*}{=} g\} \stackrel{*}{=} \{x \bullet h \stackrel{*}{=} x \bullet g\}$
- $[\![x \bullet \{h \stackrel{*}{=} \underline{g}\}]\!] \stackrel{*}{=} \{x \bullet h \stackrel{*}{=} g\}$ *(see below)*

Proof.

Immediate by definition of assertions and properties of routines.

Q.E.D.

Note that in these properties $\{x\}...$ acts as a "*filter*" on the states. This means that we only need to prove the property for states that satisfy x. The reason for this is that our notion of validity requires that any formula in $\overset{*}{Bool}$ should be proved only for states in its domain. This implies a subtle difference with Hoare logic (with only partial correctness). For example, the expression$\{false\}S\{Q\}$ is $true$ in Hoare logic for every S, since $false \Rightarrow P$ for any P. In our logic, instead, $\{false\} \blacksquare s \blacksquare Q$ is valid since its domain is empty and hence the expression is undefined everywhere. This different interpretation, however, does not compromise the equivalence of both logics.

3.7 Semantics of procedure call

In [2] the semantics of (*qualified*) routine call is defined in the following way (following Meyer we restrict ourselves to one parameter):

$$\overline{a \bullet r(x)} \stackrel{\wedge}{=} \overline{a} \blacksquare \overline{r}(\overline{x})$$

Where:

$$\overline{a} : Objects \rightarrow States \rightarrow Objects$$

and (if r is a procedure)

$$\overline{r}(\overline{x}) : Objects \rightarrow States \rightarrow States$$

or (if r is a function)

$$\overline{r}(\overline{x}) : Objects \rightarrow States \rightarrow Values$$

The operator $\blacksquare$ cannot be used since it does not have the right type. Having a correct and simple semantics for these operations is essential, since they appear frequently in object oriented programs

One possibility for the definition of its semantics is:

$$\overline{a \bullet r(x)} \stackrel{\wedge}{=} \overline{a} \bullet \overline{r}(\overline{x})$$

However, this definition has serious problems:

$[\overline{first \bullet set\_right(last)}] \blacksquare first \bullet right$
$=$ { semantics of set_right }
$[first \bullet [right := last]] \blacksquare first \bullet right$
$=$ { $P \blacksquare 2$ }
$first \bullet last$!!

Where functions $first, last$ and $set\_right()$ are as in [2]. Note that the previous equality is

not true in general, since for example expression $first \bullet last$ may be undefined. The main problem appears when a function has parameters and one tries to update attribute f in object x with the value of attribute g but coming from a different object.

In order to overcome this problem, we use the $\widehat{\ }$ operator to fix the actual parameter value and hence be able to distinguish it from the current object. This makes the semantics a little more complicated. but it is still simpler than simulating parameter passing by using assignments. The definition of (*qualified*) routine call is:

$$\overline{a \bullet r(x)} \triangleq \mathsf{function}\ o \mid \mathsf{function}\ s \mid [\overline{a} \bullet \overline{r}(\widehat{x(o)(s)})](o)(s) \tag{11}$$

Where x is the actual parameter, and $\overline{r}$ is the semantics of the routine (which will be a function, see next section)

To enhance the notation we define:

- $[\![f]\!] \triangleq [\mathsf{function}\ o \mid \mathsf{function}\ s \mid f(o)(s)]$.
- $\underline{x} \triangleq \widehat{x(o)(s)}$

Using this syntactic sugar, we can define:

$$[\![\overline{a} \bullet \overline{r}(\underline{x})]\!] \triangleq \mathsf{function}\ o \mid \mathsf{function}\ s \mid [\overline{a} \bullet \overline{r}(\widehat{x(o)(s)})](o)(s) \tag{12}$$

Some properties satisfied by $[\![-]\!]$ are:

Property 22

1. $[\![a \bullet [f := \underline{x}]]\!] \blacksquare [a \bullet f] \overset{*}{=} x$
2. $[[\![a \bullet f]\!] \blacksquare a \overset{*}{=} a] \overset{*}{\Rightarrow} [[\![a \bullet [f \blacksquare g]]\!] \overset{*}{=} [\![a \bullet f]\!] \blacksquare [\![a \bullet g]\!]]$
3. $[\![[\![f]\!]]\!] \overset{*}{=} [\![f]\!]$
4. $[\![a \bullet r]\!] \overset{*}{=} a \bullet r$ *(where r is a procedure without parameters)*

Proof.
1)
$[[\![a \bullet [f := \underline{x}]]\!] \blacksquare [a \bullet f]](o)(s)$
$=$ { *def.* $\blacksquare$ }
$[a \bullet f](o)([\![a \bullet [f := \underline{x}]]\!](o)(s))$
$=$ { *def.* $[\![-]\!]$ *and* $\underline{x}$ }
$[a \bullet f](o)([a \bullet [f := \widehat{x(o)(s)}]](o)(s))$
$=$ { *def.*$\blacksquare$ }
$[[a \bullet [f := \widehat{x(o)(s)}] \blacksquare [a \bullet f]](o)(s)$
$=$ { *prop.*$P \blacksquare 3$ }
$[a \bullet \widehat{x(o)(s)}](o)(s)$

$=$ { *prop.14.2* }
$[\widehat{x(o)(s)}](o)(s)$
$=$ { *def.* $\widehat{x(o)(s)}$ }
$x(o)(s)$
2)
This is a corollary of property $P \blacksquare 4$.
3)
Straightforward by definition.
4)
Straightforward by extensionality.
Q.E.D.

At this point, we can reconstruct the proof given in the beginning of the section, but now with the new semantic definition, which gives the expected result:

$[\overline{first \bullet set\_right(last)}] \blacksquare first \bullet right$
$=$ { semantics of set_right }
$[\![first \bullet [right := \underline{last}]]\!] \blacksquare first \bullet right$
$=$ { Prop.22.1 }
$last$

The semantics for *unqualified* routine calls, can be easily defined by replacing function a with the identity function id in the given definitions.

3.8 Procedures and functions

Given a routine:

$$r(x : T) \text{ is require } p \text{ local } a_1, ..., a_n \; body \text{ ensure } q$$

Its semantics can be defined by cases, depending on r being a procedure or a function :

- Procedure:
$$\overline{r} \triangleq \text{function } x \mid [\![body \blacksquare [a_1 := \underline{a_1}] \blacksquare ... \blacksquare [a_n := \underline{a_n}]]\!]$$
- Function:
$$\overline{r} \triangleq \text{function } x \mid [\![body \blacksquare [a_1 := \underline{a_1}] \blacksquare ... \blacksquare [a_n := \underline{a_n}] \blacksquare result]\!]$$

The final assignment statements return the old values to the local variables when the routine ends, behaving, as a stack, in a last in first out policy. Strictly speaking, in our *framework* we do not have a special syntactic category for local variables, instead we see all entities as object attributes. This abuse of notation simplifies the framework and is quite safe since in an Eiffel program the set of attribute names is disjoint with any set of local variable names.

3.9 The NewObject function.

Eiffel, as any object oriented language, provides with an object creation mechanism. The keyword `create` is used for dynamically create a new object. For example, we can consider the creation of an object of the class `LINKABLE[A]`

```
class LINKABLE[A]
creation make;
feature
    .
    .
    .
    make(v:Item) is
    do
        item := v
    end
    .
    .
    .
end
```

In Eiffel the creation operator is written as `!!`[1]for example: `!!first.make(i)`, creates and object of type `LINKABLE` and then it executes `make`. In order to formally define the semantics of object creation, we introduce a **total** function: $NewObject$.

Definition 5 *The function* $NewObject : Objects \rightarrow States \rightarrow Objects$*, takes an object* o *and a state* s *and returns an object which is not in the domain of* s*.*

This definition does not uniquely determine the function. It can be implemented in many different ways. We chose the following:

$$NewObject \stackrel{\wedge}{=} function\ o \mid function\ s \mid Max(Ran(\pi_2(s))) + 1 \tag{13}$$

In section 5 we use this function to define the semantics of `!!`. Some useful properties of $NewObject$ are:

Property 23 $[\overline{f} := NewObject]\, \blacksquare [\overline{f} \stackrel{*}{\neq} \mathbf{void}] \stackrel{*}{=} true$

Proof.

$[[f := NewObject]\, \blacksquare [f \neq \mathbf{void}]](o)(s)$
$=$ { $def.\ \blacksquare$ }
$[f \stackrel{*}{\neq} \mathbf{void}](o)([f := NewObject](o)(s))$
$=$ { $def.\ :=$, $def.NewObject$ *and* o' *is a new object* }

[1]actually, this notation is now obsolete.

$[f \overset{*}{\neq} \mathbf{void}](o)(\langle f \oplus_s o \mapsto o', \Pi_2(s))$
$=$ { $def. \neq$ }
$\overset{*}{\neg}\, [[f \overset{*}{=} \mathbf{void}](o)(\langle f \oplus_s o \mapsto o', \Pi_2(s))]$
$=$ { $def. =, def.\mathbf{void}$ }
$\overset{*}{\neg}\, [\overset{*}{false}](o)(s)$
$=$ { $logic$ }
$\overset{*}{true}\,(o)(s)$
$=$ { $\mathsf{def}.\overset{*}{true}$ }
$true$
Q.E.D.

Property 24 *For all $f, g \in Names$ it holds that: $[\overline{f} := NewObject]\, \blacksquare[f \bullet \overline{g}] \overset{*}{=} \mathbf{void}$*

Proof.

Q.E.D.

3.10 The function Current

In all object oriented languages, there is a term to refer to the current object. In Eiffel the word `current` is used. Its semantics is given in our framework by the identity function:

$$id = \mathsf{function}\ o \mid \mathsf{function}\ s \mid o$$

This function satisfies the usual properties:

Property 25 *For any function $f : Objects \nrightarrow States \nrightarrow Objects$ it holds:*

- $f \bullet id = f$
- $id \bullet f = f$

Proof.
Immediate.
Q.E.D.

4 Relating formalisms

Recently, there have been a renewed interest in formally verified programs with pointers. Some new formalisms were introduced to deal with the complexities involved (see [**?**]). The closest to our approach is the Hoare logic presented in [5], which uses a generalized version of the substitution. There is an initial contextual difference, since Bornat considers a usual imperative language, instead of an object oriented one, and hence his formalism allows direct assignment to attributes, for example:

```
p.q := x
```

which in Eiffel would be against modularity and encapsulation restrictions. The following statement will be the way to write it in Eiffel:

```
p.SetQ(x)
```

This difference is not deep, but it already involves routine calls. In both formalism, in a more or less explicit way, attributes are considered as arrays indexed by object references (or by addresses in Bornat's). For example, an assignment to component `f` of an object referred by `A` is treated as an assignment to the `A-th` component of an array `f`, and access to component `f` of object `A`, is the selection of the `A-th` component of array `f`. Hence, the array assignment laws can be used for this operations. Bornat introduces the following axioms:

$$\overline{(B.g)^{|A.f}_E \triangleq (B^{|A.f}_E).g} \quad \overline{(B.f)^{|A.f}_E \triangleq \mathsf{if}\ A = B^{A.f}_E\ \mathsf{then}\ E\ else\ (B^{|A.f}_E).f}$$

Which are obviously equivalent to the usual array assignment definitions:

$$\overline{c[j]^b_{b\oplus i\mapsto E} = c[j^b_{b\oplus i\mapsto E}]} \quad \overline{b[j]^b_{b\oplus i\mapsto E} = \mathsf{if}\ i = j^b_{b\oplus i\mapsto E}\ \mathsf{then}\ E\ else\ b[j^b_{b\oplus i\mapsto E}]}$$

Here we can see the similarity between the operation $\oplus$ used here and in Bornat's work. There is only a slight difference: our operation takes the state as an extra parameter whereas in Bornat's formalism it is left implicit. Having this in mind, we can establish the following equivalence between both formalisms:

Meyer's framework	Bornat's framework
$f \oplus_s o \mapsto \overline{E}$	$f \oplus i \mapsto E$
$a \bullet [f := g] \blacksquare h$	$h^{|a.f}_{a.g}$ (or $h^f_{f\oplus a\mapsto g[a]}$)

We can show that the array axioms hold in our formalism:

Property 26 *For all $f, g \in Names$ such that $f \neq g$ it holds that:*

$$A \bullet [\overline{f} := e] \blacksquare B \bullet \overline{g} \stackrel{*}{=} [A \bullet [\overline{f} := e] \blacksquare B] \bullet \overline{g}$$

Proof.

$[A \bullet [f := e]] \blacksquare [B \bullet g]$
$=$ { $def.\blacksquare$ }
function o | function s | $[[A \bullet [f := e]](o) \square [B \bullet g](o)](s)$
$=$ { $def.\square$ }
function o | function s | $[B \bullet g](o)([A \bullet [f := e]](o)(s))$
$=$ { $def.\bullet$ }
function o | function s | $g(B(o)([A \bullet [f := e]](o)(s)))([A \bullet [f := e]](o)(s))$
$=$ { $g \neq f$ *and* $P := 2$ }
function o | function s | $g(B(o)([A \bullet [f := e]](o)(s)))(s)$
$=$ { $def.\bullet$ }
$[[A \bullet [f := e]] \blacksquare B] \bullet g$

Q.E.D.

Property 27 *For all* $f \in Names$ *:*

1. *If* $A \bullet [\overline{f} := e] \blacksquare B \overset{*}{=} A$ *then* $A \bullet [\overline{f} := e] \blacksquare [B \bullet \overline{f}] \overset{*}{=} A \bullet e$

2. *If* $A \bullet [\overline{f} := e] \blacksquare B \overset{*}{\neq} A$ *then* $A \bullet [\overline{f} := e] \blacksquare [B \bullet \overline{f}] \overset{*}{=} [A \bullet [\overline{f} := e] \blacksquare B] \bullet \overline{f}$

Proof.

1.

$A \bullet [f := e] \blacksquare B \bullet g$
$=$ { *Def.*$\blacksquare$ }
function o | $[A \bullet [f := e]](o) \square [B \bullet f](o)$
$=$ { *def.*$\square$ }
function o |function s | $[B \bullet f](o)([A \bullet [f := e]](o)(s))$
$=$ { *def.*$\bullet$ }
function o |function s | $f(B(o)([A \bullet [f := e]](o)(s)))([A \bullet [f := e]](o)(s))$
$=$ { *def.*$\square$ }
function o |function s | $f([A \bullet [f := e](o) \square B(o)](s))([A \bullet [f := e]](o)(s))$
$=$ { *def.*$\blacksquare$ }
function o |function s | $f([A \bullet [f := e] \blacksquare B](o)(s))([A \bullet [f := e]](o)(s))$
$=$ { *Hip.* }
function o |function s | $f(A(o)(s))([A \bullet [f := e]](o)(s))$
$=$ { *def.*$\bullet$ }
function o |function s | $f(A(o)(s))([f := e](A(o)(s))(s))$
$=$ { $(P := 2)$ }
function o |function s | $e(A(o)(s))(s)$
$=$ { *def.*$\bullet$ }
$A \bullet e$

2.

$A \bullet [f := e] \blacksquare B \bullet f$
$=$ { *def.*$\blacksquare$ }
function o | $[A \bullet [f := e]](o) \,\square [B \bullet f](o)$
$=$ { *def.*$\square$ }
function o | function s | $[B \bullet f](o)(A \bullet [f := e](o)(s))$
$=$ { *def.*$\bullet$ }
function o | function s | $[B \bullet f](o)([f := e](A(o)(s))(s))$
$=$ { *def.*$\bullet$ }
function o | function s | $f(B(o)([f := e](A(o)(s))(s)))([f := e](A(o)(s))(s))$
$=$ { *Hip. and prop.* $:=$ }
function o | function s | $f(B(o)([f := e](A(o)(s))(s)))(s)$
$=$ { *def.*$\bullet$ }
function o | function s | $f(B(o)([A \bullet [f := e]](o)(s)))(s)$
$=$ { *def.*$\square$ }
function o | function s | $f([[A \bullet [f := e]](o) \,\square B(o)](s)))(s)$
$=$ { *def.*$\blacksquare$ }
function o | function s | $f([[A \bullet [f := e]] \blacksquare B](o)(s)))(s)$
$=$ { *def.*$\bullet$ }
$[A \bullet [f := e] \blacksquare B] \bullet f$

Q.E.D.

A corollary of 27 is:

Corollary 1

$$\textit{If } A \bullet [\![f := \underline{e}]\!] \blacksquare B \stackrel{*}{=} A \textit{ then } A \bullet [\![f := \underline{e}]\!] \blacksquare [B \bullet f] \stackrel{*}{=} e$$

Proof.
Immediate from theorem 27 and property 22.
Q.E.D.

Now we can prove an example from [5]:

```
{ p.b = 3 } p.a:=p { p.a.a.a.b = 3}
```

which we can write as (we do not write the operator $\bar{}$ only for notational convenience):

$$[p \bullet b \stackrel{*}{=} 3] \stackrel{*}{\Rightarrow} [\![p \bullet [a = \underline{p}]]\!] \blacksquare [p \bullet a \bullet a \bullet a \bullet b] \stackrel{*}{=} 3$$

Proof.

From property 22 it holds that:

$[\![p \bullet [a := \underline{p}]]\!] \blacksquare p \bullet a = p$
$\Rightarrow$ { Corollary 1 }
$[\![p \bullet [a := \underline{p}]]\!] \blacksquare p \bullet a \bullet a = p$
$\Rightarrow$ { Corollary 1 }
$[\![p \bullet [a := \underline{p}]]\!] \blacksquare p \bullet a \bullet a \bullet a = p$
$\Rightarrow$ { Property 26 }
$[\![p \bullet [a := \underline{p}]]\!] \blacksquare p \bullet a \bullet a \bullet a \bullet b = p \bullet b$
$\Rightarrow$ { Hyp. and transitivity of $=$ }
$[\![p \bullet [a := \underline{p}]]\!] \blacksquare p \bullet a \bullet a \bullet a \bullet b = 3$
Q.E.D.

5 Denotational semantics

In this section we present the semantics of a significant subset of Eiffel following the presentation in [2]. We make some additions and modifications. The most important is routine call, but also assignment has been redefined. This semantics is the first step towards the use of a mechanical proof checker for Eiffel program verification. For any statement e we use $\overline{e}$ to denote its semantics. The semantics is showed in three different tables:

- Basic Statements.
- Procedures and Functions.
- Proof Obligations

Basic Statements

Statement		Semantics	Type
$\overline{f}$ (attribute or local variable)	$\triangleq$	$\mathsf{function}\ o \mid \mathsf{function}\ s \mid \Pi_1(s)(f)(o)$	$[Objects \nrightarrow States \nrightarrow Values]$
$\overline{f := g}$ (assignment)	$\triangleq$	$[\overline{f} := \overline{g}]$ (see definition of := **??**)	$[Objects \nrightarrow States \nrightarrow States]$
$\overline{i; j}$ (sequential composition)	$\triangleq$	$\overline{i} \blacksquare \overline{j}$	$[Objects \nrightarrow States \nrightarrow States]$
$\overline{x.f}$ (attribute call)	$\triangleq$	$\overline{x} \bullet \overline{f}$	$[Objects \nrightarrow States \nrightarrow Values]$
$\overline{c}$ (c is a constant in *Expanded*)	$\triangleq$	$\overset{*}{c}$	$[Objects \nrightarrow States \nrightarrow Expanded]$
$\overline{e_1\ \phi\ e_2}$ (binary operator)	$\triangleq$	$\overline{e_1}\ \overset{*}{\phi}\ \overline{e_2}$	$[Objects \nrightarrow States \nrightarrow A]$
$\overline{void}$	$\triangleq$	**void**	$[Objects \nrightarrow States \nrightarrow Objects]$
$\overline{\mathsf{current}}$	$\triangleq$	id	$[Objects \nrightarrow States \nrightarrow Objects]$
$\overline{\mathsf{check}\ a}$ (assertions)	$\triangleq$	$\{\overline{a}\}$	$[Objects \nrightarrow States \nrightarrow Bool]$

Statement		**Semantics**	**Type**
$\overline{\textsf{if } b \textsf{ then } e_1 \textsf{ else } e_2}$	$\triangleq$	function o | function s | if $\overline{b}(o)(s)$ then $\overline{e_1}(o)(s)$ else $\overline{e_2}(o)(s)$	$[Objects \nrightarrow States \nrightarrow States]$
$\overline{\textsf{from } h \textsf{ until } B \textsf{ loop } g \textsf{ end}}$	$\triangleq$	$\overline{\textsf{from } h} \blacksquare \overline{\textsf{until } B \textsf{ loop } g \textsf{ end}}$	$[Objects \nrightarrow States \nrightarrow States]$
$\overline{\textsf{from } h}$	$\triangleq$	$\overline{h}$	$[Objects \nrightarrow States \nrightarrow States]$
$\overline{\textsf{until } B \textsf{ loop } g \textsf{ end}}$;	$\triangleq$	function o | function s | if $\overline{B}(o)(s)$ then s else $[\overline{g} \blacksquare \overline{\textsf{until } B \textsf{ loop } g \textsf{ end}}](o)(s)$	$[Objects \nrightarrow States \nrightarrow States]$
$\overline{!!f.g(u)}$ (creation)	$\triangleq$	$[\overline{f} := NewObject] \blacksquare \overline{f \bullet g(u)}$	$[Objects \nrightarrow States \nrightarrow States]$

Procedures

Procedure:	$r(x : T)$ is require pre local $a_1, ..., a_n$ $body$ ensure $post$	
Statement	**Semantics**	**Type**
$\overline{r}$ (Procedure r)	$\triangleq$ function x | $[\![\overline{body} \blacksquare [a_1 := \underline{a_1}] \blacksquare ... \blacksquare [a_n := \underline{a_n}]]\!]$	$[A \nrightarrow Objects \nrightarrow States \nrightarrow States]$
$\overline{r(a)}$ (procedure call)	$\triangleq$ $\overline{r}(\overline{a})$	$[Objects \nrightarrow States \nrightarrow States]$
$\overline{x.r(a)}$ (qualified procedure call)	$\triangleq$ $[\![\overline{x} \bullet \overline{r}(\underline{\overline{a}})]\!]$	$[Objects \nrightarrow States \nrightarrow States]$

Functions

Function:	$f(x : T) : B$ is require pre local $a_1, ..., a_n$ $body$ ensure $post$	
$\overline{f}$ (function f)	$\triangleq$ function x | $[\![\overline{body} \blacksquare [a_1 := \underline{a_1}] \blacksquare ... \blacksquare [a_n := \underline{a_n}] \blacksquare$ $\overline{result}]\!]$	$[A \nrightarrow Objects \nrightarrow States \nrightarrow B]$
$\overline{f(a)}$ (function call)	$\triangleq$ $\overline{f}(\overline{a})$	$[Objects \nrightarrow States \nrightarrow B]$
$\overline{x.f(a)}$ (qualified function call)	$\triangleq$ $[\![\overline{x} \bullet \overline{f}(\underline{\overline{a}})]\!]$	$[Objects \nrightarrow States \nrightarrow B]$

Additionally, certain proof obligations (which can be seen as functions of type $Objects \nrightarrow States \nrightarrow Bool$) are established for the routines:

Proof obligation	**Formula**
Class invariant inv, without occurrences of old	$[\overline{pre} \overset{*}{\wedge} \overline{inv}] \overset{*}{\Rightarrow} [\overline{body} \blacksquare \overline{inv}]$
Class invariant inv : $f\ \phi$ old g where ϕ is a binary relation	$\overline{pre} \overset{*}{\wedge} \overline{inv} \overset{*}{\Rightarrow} [\overline{body} \blacksquare \overline{f}] \overset{*}{\phi}\ g$
Routine correctness, with $post$ without occurrences of old	$\overline{pre} \overset{*}{\Rightarrow} [\overline{body} \blacksquare \overline{post}]$
Routine correctness, with $post$: $f\ \phi$ old g	$\overline{pre} \overset{*}{\Rightarrow} [\overline{body} \blacksquare \overline{f}] \overset{*}{\phi}\ g$

Furthermore, for a loop statement:

from h invariant : inv variant : v until b loop $body$ end

we establish the following proof obligations:

Proof obligation	**Formula**
Initialization, with no occurrences of old	$\overline{h} \blacksquare \overline{inv}$
Invariance , with inv without occurrences of old	$[\overline{inv} \overset{*}{\wedge} [\overset{*}{\neg} \overline{b}]] \overset{*}{\Rightarrow} [\overline{body} \blacksquare \overline{inv}]$
Initialization, with inv : $f\ \phi$ old g where ϕ is a binary relation	$[\overline{h} \blacksquare \overline{f}] \overset{*}{\phi}\ g$
Invariance, with inv : $f\ \phi$ old g	$[\overline{inv} \overset{*}{\wedge} [\overset{*}{\neg} \overline{b}]] \overset{*}{\Rightarrow} [\overline{body} \blacksquare \overline{f}] \overset{*}{\phi}\ \overline{g}$
Termination	$[\overline{body} \blacksquare \overline{v}] \overset{*}{<}\ \overline{v}$

5.1 The semantics of the loop statement

The semantics given for the loop statement $\mathsf{until}\ h\ \mathsf{loop}\ g\ \mathsf{end}$ is defined by using a recursive equation. One can take any fixed point as its solution, in particular its least fixed point which we will consider as its semantics (as, for example in [8]). We do not pursue this topic any further, since we can always use the invariance theorem in order to prove program properties. Any sematic definition which satisfies this restrictions would do (of course one should show that at least one exists, but this should be straightforward).

6 Hoare's logic

Classical Hoare's logic can be embedded in our framework in such a way that its usual rules hold:

$$\text{a) } \{P[E/x]\}x := E\{P\} \qquad \text{b) } \frac{\{P\}C_1\{Q\}, P \Rightarrow Q, \{R\}C_2\{S\}}{\{P\}C_1; C_2\{S\}}$$

$$\text{c) } \frac{\{B \wedge P\}C_1\{Q\}, \{\neg B \wedge P\}C_2\{Q\}}{\{Q\}\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \{Q\}} \qquad \text{d) } \frac{\{P \wedge B\}C\{P\}}{\{P\}\mathsf{while}\ B\ \mathsf{do}\ C\ \{\neg B \wedge P\}}$$

Each rule can be rewritten in the following way:

Property 28 (a) *For all predicates P, it holds that:*

$$\{\overline{P[E/x]}\} \blacksquare \overline{[x := E]} \blacksquare \overline{P}$$

Proof.
By induction on the structure of P.
Q.E.D.

Expression $P[E/x]$ is the result of replacing x by E in P.

Property 29 (b) *If: $\{P\} \blacksquare C_1 \blacksquare Q$, $Q \stackrel{*}{\Rightarrow} R$ and $\{R\} \blacksquare C_2 \blacksquare S$ then: $\{P\} \blacksquare C_1 \blacksquare C_2 \blacksquare S$.*
Proof.

Q.E.D.

Property 30 (c) *The following property holds:*

$$\{P\} \blacksquare \overline{\mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2} \blacksquare Q$$

if and only if:

- $\{P \stackrel{*}{\wedge} B\} \blacksquare S_1 \blacksquare Q$
- $\{P \stackrel{*}{\wedge} [\stackrel{*}{\neg} B]\} \blacksquare S_2 \blacksquare Q$

Proof.
Immediate from the semantics of conditional statement.
Q.E.D.

Property 31 (d) *If the following holds:*

1. $[P \overset{*}{\wedge} \overline{B}] \overset{*}{\Rightarrow} Q$
2. $\{P \overset{*}{\wedge} [\overset{*}{\neg} \overline{B}]\} \blacksquare \overline{S} \blacksquare P$

then the following holds as well: $\{P\} \blacksquare \overline{\mathsf{until}\ B\ \mathsf{loop}\ S\ \mathsf{end}} \blacksquare Q$
Proof.
by straightforward fixed point induction.
Q.E.D.

Furthermore, if there is a function $t : Objects \nrightarrow States \nrightarrow Int$ which satisfies:

- $P \overset{*}{\wedge} [\overset{*}{\neg} B] \overset{*}{\Rightarrow} t \leq 0$
- $[P \overset{*}{\wedge} [\overset{*}{\neg} B] \overset{*}{\wedge} [t = c]] \blacksquare S \blacksquare \{t < c\}$

then the *loop* terminates.

7 Sequences as models

In this section we apply the theory to verify some properties of the class `LINKED_LIST`. This class implement linked lists.

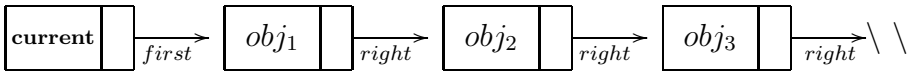

Figure 2: Example of LINKED_LIST.

Every object of the class has a pointer to the first element (a `LINKABLE` object), and every one of them has a pointer to the next (`right`). The abstract model for the linked lists is the model of sequences. For example, the model associated with figure 2 is:

$$\langle obj_1, obj_2, obj_3 \rangle$$

In order to prove class correctness of `LINKED_LIST` we express assertions using the abstract model. The properties that the abstract model satisfies are all well known. We use a functional programming style for describing the type $SEQ[A]$ of sequences of objects of type A. We consider the following operations:

- $\rhd$: $A \to SEQ[A] \to SEQ[A]$, cons.
- $\lhd$: $A \to SEQ[A] \to SEQ[A]$, snoc (adds an element at the end).
- $+\!\!+ : SEQ[A] \to SEQ[A] \to SEQ[A]$, concatenation.
- $hd : SEQ[A] \to A$, head.
- $tl : SEQ[A] \to SEQ[A]$, tail.
- $rev : SEQ[A] \to SEQ[A]$, reverse of a sequence.
- $\langle\rangle$, empty sequence.
- $\langle - \rangle : A \to SEQ[A]$, one-point sequence.

The formal definitions and properties are standard and can be looked for in any text of functional programming, for example [6]. Furthermore, we will consider the *pointwise* extension of the operations, denoted as usual by $\overset{*}{\rhd}, \overset{*}{\lhd}, \overset{*}{+\!\!+}, \overset{*}{hd}, \overset{*}{tl}, \overset{*}{rev}$. All the properties are preserved and by theorem 1 they distribute with respect to the operator $\blacksquare$, for example, in the case of concatenation:

$$p \blacksquare [s_1 \overset{*}{+\!\!+} s_2] = [p \blacksquare s_1] \overset{*}{+\!\!+} [p \blacksquare s_2]$$

We call this property $\mathsf{Dist}.\ \overset{*}{+\!\!+}, \blacksquare$.

In this work we will only consider finite lists without repeated elements. To ensure this property we introduce a class invariant based on the following inductive predicate which expresses that we can reach an object from another in exactly n f-steps, where $f \in Names$. Its formal definition is the following :

$$[p \underset{0}{\overset{f}{\rightsquigarrow}} q] \triangleq [p \overset{*}{=} q]$$
$$[p \underset{n+1}{\overset{f}{\rightsquigarrow}} q] \triangleq [p \overset{*}{\neq} q] \overset{*}{\wedge} [[p \bullet f] \underset{n}{\overset{f}{\rightsquigarrow}} q]$$

This predicate is well defined, even when $p \overset{*}{=} \mathbf{void}$ since the second equation is correct given that $\mathbf{void} \bullet f = \mathbf{void}$. Some useful properties of $\underset{n}{\overset{f}{\rightsquigarrow}}$ are:

Property 32 *Let $p, q : Objects \nrightarrow States \nrightarrow States$ and $f \in Names$:*

- $[p \underset{0}{\overset{f}{\rightsquigarrow}} p] \overset{*}{\equiv} true$
- $[p \underset{1}{\overset{f}{\rightsquigarrow}} p \bullet f] \overset{*}{\equiv} true$
- $\overset{*}{\forall}\, n : \mathbb{N} \bullet [q \neq \mathbf{void}] \overset{*}{\Rightarrow} \overset{*}{\neg} [\mathbf{void} \underset{n}{\overset{f}{\rightsquigarrow}} q]$

- $\overset{*}{\forall}\, n, m : \mathbb{N} \bullet [p \underset{n}{\overset{f}{\rightsquigarrow}} q] \overset{*}{\wedge} [q \underset{m}{\overset{f}{\rightsquigarrow}} t] \overset{*}{\equiv} [p \underset{n+m}{\overset{f}{\rightsquigarrow}} t]$

Proof.

The first two are immediate by definition. The other two by induction on n

Q.E.D.

In order to verify that this predicate is a class invariant we need to see how it interacts with assignments. Some important properties are:

Property 33 *Let $p, q, f \in Names$ and $x : Objects \nrightarrow States \nrightarrow Objects$:*

- $[p \neq q] \overset{*}{\wedge} [p \neq f] \overset{*}{\Rightarrow} [p := x] \blacksquare [p \underset{n}{\overset{f}{\rightsquigarrow}} q] \overset{*}{\equiv} [x \underset{n}{\overset{f}{\rightsquigarrow}} q]$

- $[q \neq p] \overset{*}{\wedge} [q \neq f] \overset{*}{\Rightarrow} [p := x] \blacksquare [p \underset{n}{\overset{f}{\rightsquigarrow}} q] \overset{*}{\equiv} [p \underset{n}{\overset{f}{\rightsquigarrow}} x]$

Proof.

Induction on n.

Q.E.D.

The model proposed in [1] for `LINKED_LIST` is rather different. An operator of reflexive and transitive closure ($(-)^{**}$) is introduced, and with it the expression $first \bullet right^{**}$ is used to denote all elements of the list including the first. This definition, however, introduces some problems, for example when $[first \overset{*}{=} \textbf{void}]$ its value is undefined. Although it can be corrected we chose to use a different approach following [5]. The sequence $p \overset{f}{\Rightarrow} q$ will contain all elements beginning in p until q is reached (q itself is excluded). Obviously then, the expression $[p \overset{right}{\Rightarrow} \textbf{void}]$ denotes all objects in the `LINKED_LIST` (it may be an infinite list if there are cycles). The formal definition of $p \overset{f}{\Rightarrow} q$ is:

$$\begin{array}{ll} \mathsf{function}\ o \mid \mathsf{function}\ s \mid & \mathsf{if}\ [[p \overset{*}{=} q] \overset{*}{\vee} [p \overset{*}{=} \textbf{void}]](o)(s)\ \mathit{then}\ \langle\rangle \\ & \mathsf{else}\ \langle p(o)(s)\rangle +\!\!+ [p \bullet f \overset{f}{\Rightarrow} q](o)(s) \end{array}$$

An example is showed in figure 3.

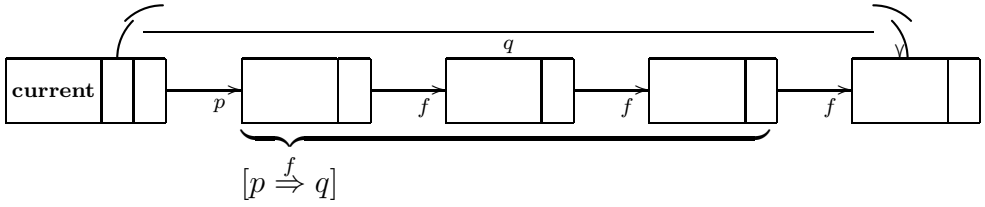

Figure 3: Operator $\overset{f}{\Rightarrow}$

When $[first \overset{*}{=} \mathbf{void}]$ it follows that $[first \overset{f}{\Rightarrow} \mathbf{void}](o)(s) = \langle\rangle$, for every $o \in Objects, s \in States$. All the elements of $\overset{f}{\Rightarrow}$ are objects instead of references. This definition may be undefined when it is not possible to reach q from p. We require then as class invariant the following predicate, which can be interpreted as saying that the list is finite.

$$\overset{*}{\exists}\, n : \mathbb{N} \bullet [p \underset{n}{\overset{f}{\rightsquigarrow}} q] \tag{14}$$

The following property shows that when 14 holds, then $\overset{f}{\Rightarrow}$ is well defined.

Property 34 $\overset{*}{\forall}\, n : \mathbb{N} \bullet [p \underset{n}{\overset{f}{\rightsquigarrow}} q] \overset{*}{\Rightarrow} [[p \overset{f}{\Rightarrow} q] \,\square\, lenght \overset{*}{=} \overset{*}{n}]$

Proof..

Base case-

If $n = 0$ then $[p \overset{*}{=} q]$ which implies $[p \overset{f}{\Rightarrow} q](o)(s) = \langle\rangle$, and hence $[p \overset{f}{\Rightarrow} q] \,\square\, lenght \overset{*}{=} 0$.

Induction hypothesis- Assume it holds for a given n.

Inductive case-

Assume that $p \underset{n+1}{\overset{f}{\rightsquigarrow}} q$ (Sp.Hyp.):

$[[p \overset{f}{\Rightarrow} q] \,\square\, lenght](o)(s)$
$\equiv$ { *def.*$\square$ }
$lenght([p \overset{f}{\Rightarrow} q](o)(s))$
$\equiv$ { *Sp.Hyp. and def.*$\overset{f}{\Rightarrow}$ }
$lenght(\langle p(o)(s)\rangle +\!\!+ [p \bullet f \overset{f}{\Rightarrow} q](o)(s))$
$\equiv$ { *def.lenght* }
$lenght([p \bullet f \overset{f}{\Rightarrow} q](o)(s)) + 1$
$\equiv$ { *Sp.Hyp. and I.H.* }
$n + 1$
Q.E.D.

The following are some useful properties of $p \overset{f}{\Rightarrow} q$.

Property 35 . *Let $p, q, f \in Names$ pairwise different, and $x, h, g \in Objects \nrightarrow States \nrightarrow Objects$. If $\overset{*}{\exists}\, n : \mathbb{N} \bullet [p \underset{n}{\overset{f}{\rightsquigarrow}} q]$ then it holds that:*

1. $[h := q] \blacksquare [p \overset{f}{\Rightarrow} h] \overset{*}{=} [p \overset{f}{\Rightarrow} q]$
2. $[h := p] \blacksquare [h \overset{f}{\Rightarrow} q] \overset{*}{=} [p \overset{f}{\Rightarrow} q]$
3. *If* $x \overset{*}{\in} [p \overset{f}{\Rightarrow} q]$ *then* $[p \overset{f}{\Rightarrow} q] \overset{*}{=} [p \overset{f}{\Rightarrow} x] \overset{*}{+\!\!+} [x \overset{f}{\Rightarrow} q]$

4. $[[q \bullet f \overset{*}{\neq} \mathbf{void}] \overset{*}{\wedge} [q \overset{*}{\neq} h]] \overset{*}{\Rightarrow} [[g \overset{f}{\Rightarrow} q] \overset{*}{+\!\!+} [q \overset{f}{\Rightarrow} h]] \overset{*}{=} [[g \overset{f}{\Rightarrow} q \bullet f] \overset{*}{+\!\!+} [q \bullet f \overset{f}{\Rightarrow} h]]$

5. $[p \overset{*}{\neq} q] \overset{*}{\Rightarrow} [p \overset{f}{\Rightarrow} q \overset{*}{=} p \overset{f}{\Rightarrow} p \bullet f \overset{*}{+\!\!+} p \bullet f \overset{f}{\Rightarrow} q]$

6. $[p := q] \blacksquare [p \overset{f}{\Rightarrow} p \bullet f] \overset{*}{=} [q \overset{f}{\Rightarrow} q \bullet f]$

7. $[p \overset{*}{\neq} \mathbf{void}] \overset{*}{\Rightarrow} [\![p \bullet [f := \underline{h}]]\!] \blacksquare [p \bullet f \overset{f}{\Rightarrow} q] \overset{*}{=} h \overset{f}{\Rightarrow} q$

8. $[p \overset{*}{=} g] \overset{*}{\Rightarrow} [[p \overset{f}{\Rightarrow} q] \overset{*}{=} [g \overset{f}{\Rightarrow} q]]$

9. $[q \overset{*}{=} g] \overset{*}{\Rightarrow} [[p \overset{f}{\Rightarrow} q] \overset{*}{=} [p \overset{f}{\Rightarrow} g]]$

Proof.

By induction on n. We only proof the first one, the other proofs are similar.

1)

Base Case-

if $\overset{*}{\exists}\, n : \mathbb{N} \bullet [p \underset{0}{\overset{f}{\rightsquigarrow}} q]$ *then* $[p \overset{*}{=} q]$ *(*) and therefore by property* **??***:* $[p \overset{*}{=} q](o)([q := h](o)(s))$ *(**), for every object o and state s. We apply the left hand side to an arbitrary object and an arbitrary state:*

$[[h := q] \blacksquare [p \overset{f}{\Rightarrow} h]](o)(s)$

$=$ { *Def.*$\overset{f}{\Rightarrow}$ *and (**)* }

$\langle\rangle$

$=$ { *Def.*$\overset{f}{\Rightarrow}$ *and (*)* }

$[p \overset{f}{\Rightarrow} q](o)(s)$

Induction Hypothesis- Suppose that the property holds for n.

Inductive Case- Suppose that $\overset{*}{\exists}\, n : \mathbb{N} \bullet [p \underset{n+1}{\overset{f}{\rightsquigarrow}} q]$*(***) holds then:*

$[h := q] \blacksquare [p \overset{f}{\Rightarrow} h](o)(s)$

$=$ { *Def.*$\overset{f}{\Rightarrow}$ *and (***)* }

$\langle p(o)([h := q](o)(s)) \rangle \overset{*}{+\!\!+} [[p \bullet f] \overset{f}{\Rightarrow} h](o)([h := q](o)(s))$

$=$ { *Prop.P:=2 and Ind.Hyp.* }

$\langle p(o)(s) \rangle \overset{*}{+\!\!+} [[p \bullet f] \overset{f}{\Rightarrow} q](o)(s)$

$=$ { *Def.*$\overset{f}{\Rightarrow}$ *and (***)* }

$p \overset{f}{\Rightarrow} q$

Q.E.D.

Additionally, we will establish some properties of $\overset{f}{\Rightarrow}$ when combined with list operations.

Property 36 . *Let $p, q, f \in Names$ pairwise different. If $\overset{*}{\exists}\, n : \mathbb{N} \bullet [p \overset{f}{\underset{n}{\rightsquigarrow}} q]$ then:*

1. $[p \overset{*}{\neq} \mathbf{void}] \overset{*}{\wedge} [p \overset{*}{\neq} q] \overset{*}{\Rightarrow} [p \overset{f}{\Rightarrow} q] \,\square\, hd \overset{*}{=} p$
2. $[p \overset{*}{\neq} q] \overset{*}{\wedge} [q \overset{*}{\neq} \mathbf{void}] \overset{*}{\Rightarrow} [p \overset{f}{\Rightarrow} q \overset{*}{+\!\!+} q \overset{f}{\Rightarrow} q \bullet f] \,\square\, lst \overset{*}{=} q$
3. $[p \overset{*}{\neq} \mathbf{void}] \overset{*}{\Rightarrow} [p \overset{f}{\Rightarrow} p \bullet f \overset{*}{+\!\!+} p \bullet f \overset{f}{\Rightarrow} q] \,\square\, tl \overset{*}{=} [p \bullet f \overset{f}{\Rightarrow} q]$
4. $[p \overset{f}{\Rightarrow} p] \,\square\, is\_empty \overset{*}{=} true$
5. $p \overset{f}{\Rightarrow} p \overset{*}{=} \langle\rangle$

Proof.
By induction on n.
Q.E.D.

We can now define the model for class `LINKED_LIST[A]` as follows:

$$Model = [first \overset{right}{\Rightarrow} \mathbf{void}]$$

And the class invariant is:

$$no\_cycles : \overset{*}{\exists}\, n : \mathbb{N} \bullet [first \overset{right}{\underset{n}{\rightsquigarrow}} \mathbf{void}]$$

This can be understood as saying that *model* represents a finite list. Moreover, the invariant ensures that we can reach **void** from $first$, and hence there are no cycles.

In the following sections we define the assertions on the abstract model by using the extended operators. for example,

$$old\ model = [first \overset{right}{\Rightarrow} last] \overset{*}{+\!\!+} [last \overset{right}{\Rightarrow} void]$$

To illustrate these properties, we show the correctness of the following initialization method for `LINKED_LIST`.

```
Make is
do
        first = void
ensure
        empty: model □ is_empty
end
```

We show how to prove its post-condition:

$[[first := \mathbf{void}] \blacksquare model] \,\square\, is\_empty$
$=$ { Def. Model }

$[[first := \mathbf{void}]\blacksquare[first \overset{right}{\Rightarrow} \mathbf{void}]]\,\square is\_empty$
$=$ { Prop. 35.2 }
$[\mathbf{void} \overset{right}{\Rightarrow} \mathbf{void}]\,\square is\_empty$
$=$ { Prop.36.4 }
$true$

We remark that the proof would have not been possible with the former definition $Model = first \bullet right^{**}$.

8 Spatial Separation

One of the main difficulties when reasoning with pointer structures like $first \overset{f}{\Rightarrow} \mathbf{void}$ is the fact that an assignment on one component can have unpredictable effects on the rest of the structure. The property of **spatial separation** allows to express the fact that certain assignments do not affect certains parts of the pointer structure. This is needed for practical proof of programs. This property can be expressed in this framework as follows:

Theorem 3 (Spatial Sep.) *Let $x : Objects \nrightarrow States \nrightarrow Objects$ and $p, f, q \in Names$ pairwise different. If $\overset{*}{\exists}\, n : \mathbb{N} \bullet [p \underset{n}{\overset{f}{\rightsquigarrow}} q]$ then:*

$$[x \overset{*}{\neq} \mathsf{void}] \overset{*}{\wedge} [x \overset{*}{\notin} [p \overset{f}{\Rightarrow} q]] \overset{*}{\Rightarrow} [[x \bullet [f := y]]\blacksquare[p \overset{f}{\Rightarrow} q] \overset{*}{=} [p \overset{f}{\Rightarrow} q]]$$

Proof.

The proof is by induction on n.

Base Case-

if $n = 0$ then $[p \overset{*}{=} q]$ (*), and therefore by lemma 1: $x \bullet [f := y]\blacksquare p \overset{*}{=} q$ (**). On the other hand:

$[x \bullet [f := y]]\blacksquare[p \overset{f}{\Rightarrow} q]$
$=$ { *by (**) and def.*$\overset{f}{\Rightarrow}$ }
$\overset{*}{\langle\rangle}$
$=$ { *by (*) and def.*$\overset{f}{\Rightarrow}$ }
$[p \overset{f}{\Rightarrow} q]$

Inductive Hypothesis- The property holds for an arbitrary n.

Inductive Case-

Suppose that $\overset{*}{\exists}\, n : \mathbb{N} \bullet [p \underset{n+1}{\overset{f}{\rightsquigarrow}} q]$ holds (*):

$[[x \bullet [f := y]]\blacksquare[p \overset{f}{\Rightarrow} q]$
$=$ { *by * and* $\overset{f}{\Rightarrow}$ *definition* }

$[[x \bullet [f := y]] \blacksquare p \overset{*}{\triangleright} [p \bullet f \overset{f}{\Rightarrow} q]$

$=$ { *dist.*$\blacksquare$,$\overset{*}{\triangleright}$ }

$[[[x \bullet [f := y]] \blacksquare p] \overset{*}{\triangleright} [[[x \bullet [f := y]] \blacksquare p \bullet f \overset{f}{\Rightarrow} q]$

$=$ { *I.H. and lemma 1* }

$p \overset{*}{\triangleright} [p \bullet f \overset{f}{\Rightarrow} q]$

$=$ { *def.*$\overset{f}{\Rightarrow}$ }

$p \overset{f}{\Rightarrow} q$

Q.E.D.

Where:

Lemma 1 *Let $p, f \in Names$: If $p \neq f$ then $[x \neq \mathsf{void}] \overset{*}{\Rightarrow} [[x \bullet [f := y] \blacksquare p] \overset{*}{=} p]$*

Proof.

Straightforward by properties of $:=$.

Q.E.D.

Intuitively, a change to a reference outside the model will not affect it. An interesting corollary of this property is that creation methods do not affect the model, i.e. if $x \in Names$ y $make(a : T)$ is a creation method for x then:

$$[x \notin [p \overset{f}{\Rightarrow} q]] \overset{*}{\Rightarrow} [[!!\overline{x} \bullet make(a)] \blacksquare [p \overset{f}{\Rightarrow} q] \overset{*}{=} [p \overset{f}{\Rightarrow} q]] \tag{15}$$

This property can be proven directly from the definition of !!, since after execution of *make*, x "points to" a fresh object.

An interesting point about spatial separation is that it can be used for any inductively defined structure (e.g. trees). Spatial separation between a data structure and an object x ca always be expressed as $o \notin S$ where S is the set of objects involved in the given data structure.

The following properties are useful to reason about acyclic lists:

Property 37 (Spatial Sep. Prop.) *Let $x, p, q, f \in Names$ pairwise different. If $\overset{*}{\exists}\ n : \mathbb{N} \bullet [p \underset{n}{\overset{f}{\rightsquigarrow}} q]$ then:*

1. $[[x := h] \blacksquare [p \overset{f}{\Rightarrow} q]] \overset{*}{=} [p \overset{f}{\Rightarrow} q]$
2. $q \bullet [f := h] \blacksquare [p \overset{f}{\Rightarrow} q] \overset{*}{=} [p \overset{f}{\Rightarrow} q]$
3. $p \bullet [f := h] \blacksquare [p \overset{f}{\Rightarrow} p \bullet f] \overset{*}{=} p \overset{f}{\Rightarrow} p \bullet f$

Proof.

Q.E.D.

In the following sections we use the predicate $\pitchfork$ which expresses the disjointness of two lists, in the same way that is done in [5]. Its definition is:

$\langle\rangle \pitchfork ys = true$
$x \rhd xs \pitchfork ys = (x \notin ys) \wedge (xs \pitchfork ys)$

Obviously we use its pointwise extension $\overset{*}{\pitchfork}$. Some properties of $\overset{*}{\pitchfork}$ are the following.

Property 38 (P.$\overset{*}{\pitchfork}$) *Let $p, f, q, x, y \in Names$ pairwise different, if $\overset{*}{\exists}\ n : \mathbb{N} \bullet [p \overset{f}{\underset{n}{\rightsquigarrow}} q]$ and $\overset{*}{\exists}\ n : \mathbb{N} \bullet [x \overset{f}{\underset{n}{\rightsquigarrow}} y]$ then:*

- $[p \overset{f}{\Rightarrow} q] \overset{*}{\pitchfork} [x \overset{f}{\Rightarrow} y] \overset{*}{\Rightarrow} p \overset{*}{\notin} [x \overset{f}{\Rightarrow} y]$
- $[p \overset{f}{\Rightarrow} p \bullet f] \overset{*}{\pitchfork} [p \bullet f \overset{f}{\Rightarrow} q]$
- $[[p \overset{f}{\Rightarrow} q] \overset{*}{\pitchfork} [x \overset{f}{\Rightarrow} y]] \overset{*}{\Rightarrow} [[p \overset{f}{\Rightarrow} q] \overset{*}{\pitchfork} [x \bullet f \overset{f}{\Rightarrow} y]]$

Proof.

Q.E.D.

9 An application: LINKED_LIST

In this section we apply the theory to the class $LINKED\_LIST$ which has been already introduced in previous sections. We consider four operations:

- `Make`, creation.
- `last`, last element of a linked list.
- `add_item`, adds an item to the front of a list.
- `remove_front`, removes the head of a list.

We must consider also the class `LINKABLE`.

```
Class LINKABLE[G]
    create
        make
    feature
        make(v:item) is
        do
            item:=v;
        end
```

```
    feature
        put_right(other:like current) is
        do
            right:=other
        ensure
            chained:right = other
        end
end
```

Furthermore, class LINKED_LIST can be defined as follows:

```
Class LINKED_LIST[G]
    create
        make
    feature
        first: LINKABLE  -- head of the list
        last: like first is -- return the last item
            local
                q, p:like first
            require
                not_void:  [first ≠ void]
            do
                from
                    q:=first
                    p:=first.right
                invariant:
                    loop_inv: [old model = [first ⇒(right) q] #(*) [q ⇒(right) void]]
                loop
                    q:=p
                    p:=p.right
                until
                    q.right = void
                end
               result:=q
            ensure
                is_last: [result = model □ lst]
            end

        add_item(i:item) is -- add a new item
            local
                aux:LINKABLE
            do
                !!aux.make(i)
                aux.put_right(first)
```

```
            first:=aux
        ensure
            it_added:  [aux = model □ hd]
            model_tail:  [old model = model □ tl]
        end

    remove_front is
        -- remove first element of list
        require
            not_empty: [first ≠ void]
        do
            first:=first.right
        ensure
            removed:  [model = old model □ tl]
        end

    make is
        do
            first:=void
        ensure
            created:  [model □ is_empty]
        end

invariant:
    no_cycles:  [∃* n : ℕ • [first ⇝(right, n) void]]
```

9.1 Correctness of last

We prove the following items:

1. loop invariant.
   (a) Initialization: $[q := first]\,\blacksquare\,\overline{loop\_inv}$
   (b) Invariance: $[\overline{loop\_inv} \overset{*}{\wedge} [q \bullet right \neq \textbf{void}]] \overset{*}{\Rightarrow} [q := q \bullet right]\,\blacksquare\,\overline{loop\_inv}$
2. PostCondition: $\{\overline{loop\_inv} \overset{*}{\wedge} q \bullet right = \textbf{void}\}\,\blacksquare[result := q]\,\blacksquare[result = model \,\square\, lst]$

We first prove the invariant and then use it to show the correctness of the postcondition.

9.1.1 Invariant proof

Initialization:

$[q := first]\,\blacksquare[[first \overset{right}{\Rightarrow} q] \overset{*}{+} [q \overset{right}{\Rightarrow} \mathbf{void}]] = model$
$=$ { Prop.35.1, Prop.35.2, and Dist.$\blacksquare, \overset{*}{+}$ }
$[first \overset{right}{\Rightarrow} first \overset{*}{+} first \overset{right}{\Rightarrow} \mathbf{void}] = model$
$=$ { Prop.35.4 }
$first \overset{right}{\Rightarrow} \mathbf{void} = model$
$=$ { reflex.$=$ }
$true$

Invariance:

$[[q := q \bullet right]\,\blacksquare[[first \overset{right}{\Rightarrow} q] \overset{*}{+} [q \overset{right}{\Rightarrow} \mathbf{void}]] = model$
$=$ { Dist.$\blacksquare, \overset{*}{+}$ and properties 35.1 35.2 }
$[first \overset{right}{\Rightarrow} q \bullet right] \overset{*}{+} [q \bullet right \overset{right}{\Rightarrow} \mathbf{void}]$
$=$ { Prop.35.10 and hyp. $(q \bullet right \neq \mathbf{void})$ }
$[first \overset{right}{\Rightarrow} q] \overset{*}{+} [q \overset{right}{\Rightarrow} \mathbf{void}]$
$=$ { hyp. }
$model$

Q.E.D.

9.1.2 Postcondition correctness

Assume the antecedent:

$$[q \bullet right = \mathbf{void}] \overset{*}{\wedge} [model = [first \overset{right}{\Rightarrow} q] \overset{*}{+} [q \overset{right}{\Rightarrow} \mathbf{void}]]$$

we prove the consequent:

$[result := q]\,\blacksquare[result = model \,\square\, lst]$
$=$ { Dist.$\blacksquare$,$=$. and properties 37 }
$[q = model \,\square\, lst]$
$=$ { Hip. }
$[q = [first \overset{right}{\Rightarrow} q] \overset{*}{+} [q \overset{right}{\Rightarrow} \mathbf{void}] \,\square\, lst]$
$=$ { Hip.$(q \bullet right =$ void) y Prop.35.8 }
$[q = [[first \overset{right}{\Rightarrow} q] \overset{*}{+} [q \overset{right}{\Rightarrow} q \bullet right]] \,\square\, lst]$
$=$ { $q \neq void$ y Prop.36.2 }
$[q = q]$
$=$ { reflex.$=$ }

$true$
Q.E.D.

9.2 Correctness of add_item

The following items are the proof obligations:

1. Postcondition it_added: $\overline{body} \blacksquare \overline{it\_added}$
2. Postcondition model_tl: $\overline{body} \blacksquare \overline{model\_tl}$
3. Class Invariant: $[no\_cycles] \stackrel{*}{\Rightarrow} [\overline{body} \blacksquare no\_cycles]$

Where *body* is the routine body.

9.2.1 Postcondition correctness

it_added correctness:

If we prove:

$$[\overline{!!aux \bullet make(i)} \blacksquare \overline{aux \bullet put\_right(first)} \blacksquare [first := aux] \blacksquare model] \Box hd = aux$$

then, by definition of $aux \bullet make$ and $aux \bullet put\_right$, it holds that:

$$\overline{!!aux \bullet make(i)} \blacksquare \overline{aux \bullet put\_right(first)} \blacksquare [aux \neq void \stackrel{*}{\wedge} aux \bullet right = first]$$

i.e., if we prove:

$$[aux \neq void \stackrel{*}{\wedge} aux \bullet right = first] \stackrel{*}{\Rightarrow} [first := aux \blacksquare model] \Box hd = aux$$

then by theorem 2 the correctness will be proved. We suppose the antecedent and prove the consequent:

$[[first := aux] \blacksquare [first \stackrel{right}{\Rightarrow} void]] \Box hd$
$=$ { Prop. 35.2 }
$[aux \stackrel{right}{\Rightarrow} void] \Box hd$
$=$ { Hyp. and Prop. }
$[[aux \stackrel{right}{\Rightarrow} aux \bullet right] \stackrel{*}{+} [aux \bullet right \stackrel{right}{\Rightarrow} \mathbf{void}]] \Box hd$
$=$ { $[aux \neq \mathbf{void}]$ and prop.36.1 }
aux
Q.E.D.

Model_tl correctness:

Similar to the previous proof.

9.2.2 Class invariant correctness

We have to prove:

$$[no\_cycles] \stackrel{*}{\Rightarrow} [\overline{body} \blacksquare no\_cycles]$$

$[\overline{body} \blacksquare no\_cycles]$
$\equiv$ { def.body and def. no_cycles }
$[\overline{!!aux \bullet make(i)} \blacksquare \overline{aux \bullet put\_right(first)} \blacksquare [first := aux] \blacksquare \stackrel{*}{\exists} n : \mathbb{N} \bullet [first \stackrel{right}{\underset{n}{\rightsquigarrow}} \mathbf{void}]]$
$\equiv$ { Dist.$\blacksquare$, $\stackrel{*}{\exists}$ and properties 33 }
$[\overline{!!aux \bullet make(i)} \blacksquare \overline{aux \bullet put\_right(first)} \blacksquare \stackrel{*}{\exists} n : \mathbb{N} \bullet [aux \stackrel{right}{\underset{n}{\rightsquigarrow}} \mathbf{void}]]$
$\Leftarrow$ { logic }
$[\overline{!!aux \bullet make(i)} \blacksquare \overline{aux \bullet put\_right(first)} \blacksquare \stackrel{*}{\exists} n : \mathbb{N} \bullet [aux \stackrel{right}{\underset{n+1}{\rightsquigarrow}} \mathbf{void}]]$
$\equiv$ { Def. $\stackrel{right}{\underset{n}{\rightsquigarrow}}$ and $aux \neq \mathbf{void}$ }
$[\overline{!!aux \bullet make(i)} \blacksquare \overline{aux \bullet put\_right(first)} \blacksquare \stackrel{*}{\exists} n : \mathbb{N} \bullet [aux \bullet right \stackrel{right}{\underset{n}{\rightsquigarrow}} \mathbf{void}]]$
$\equiv$ { def. put_right y prop. ... }
$[\overline{!!aux \bullet make(i)} \blacksquare \stackrel{*}{\exists} n : \mathbb{N} \bullet [first \stackrel{right}{\underset{n}{\rightsquigarrow}} \mathbf{void}]]$
$\equiv$ { spatial separation }
$\stackrel{*}{\exists} n : \mathbb{N} \bullet [first \stackrel{right}{\underset{n}{\rightsquigarrow}} \mathbf{void}]$
$\equiv$ { hip. }
$true$

Q.E.D.

9.3 Correctness of remove_front

We have to prove the following items:

- Method correctness: $[first \neq \mathbf{void}] \stackrel{*}{\Rightarrow} [[first := first \bullet right] \square model = [oldmodel] \square tl]$
- Class invariant: $[no\_cycles] \stackrel{*}{\Rightarrow} [[first := first \bullet right] \blacksquare no\_cycles]$

9.3.1 Method correctness

$[first := first \bullet right] \square model]$
$=$ { def.$model$ }
$[first := first \bullet right] \square [first \stackrel{right}{\Rightarrow} void]$
$=$ { Prop.35.2 }

$[first \bullet right \overset{right}{\Rightarrow} void]$
$=$ { Hip ($first \neq \mathbf{void}$) }
$[[first \overset{right}{\Rightarrow} first \bullet right] \overset{*}{+} [first \bullet right \overset{right}{\Rightarrow} \mathbf{void}]] \,\square\, tl$
$=$ { Prop.36.3 }
$[first \overset{right}{\Rightarrow} \mathbf{void}] \,\square\, tl$
$=$ { def. $model$ }
$model \,\square\, tl$
Q.E.D.

9.3.2 Class invariant

$[first := first \bullet right] \blacksquare no\_cycles$
$\equiv$ { def. no_cycles }
$[first := first \bullet right] \blacksquare [\overset{*}{\exists}\, n : \mathbb{N} \bullet [first \overset{right}{\underset{n}{\rightsquigarrow}} \mathbf{void}]]$
$\equiv$ { prop. 33.1 }
$\overset{*}{\exists}\, n : \mathbb{N} \bullet [first \bullet right \overset{right}{\underset{n}{\rightsquigarrow}} \mathbf{void}]$
$\Leftarrow$ { def. $\overset{right}{\underset{n}{\rightsquigarrow}}$ }
$[first \neq \mathbf{void}] \overset{*}{\wedge} [\overset{*}{\exists}\, n : \mathbb{N} \bullet [first \overset{right}{\underset{n}{\rightsquigarrow}} \mathbf{void}]]$

Q.E.D.

9.4 Correctness of make

The proof obligations are:

- Method correctness. (see section 7)
- Class invariant: $[first := void] \blacksquare [\overset{*}{\exists}\, n : \mathbb{N} \bullet [first \overset{right}{\underset{n}{\rightsquigarrow}} \mathbf{void}]]$

9.4.1 Class invariant

$[[first := void] \blacksquare [\overset{*}{\exists}\, n : \mathbb{N} \bullet [first \overset{right}{\underset{n}{\rightsquigarrow}} \mathbf{void}]]$
$\equiv$ { Prop. 33.1 }
$[\overset{*}{\exists}\, n : \mathbb{N} \bullet [\mathbf{void} \overset{right}{\underset{n}{\rightsquigarrow}} \mathbf{void}]]$
$\equiv$ { def. $\overset{right}{\underset{n}{\rightsquigarrow}}$ and $n = 0$ }
$true$

Q.E.D.

10 A last example: the **reverse** algorithm

Several papers on references (see: [5],[2],[13]) show an algorithm for reversing lists as an example of moderate complexity. In this paper we take the version of this algorithm showed in [2] (with some corrections) and prove the correctness of the loop invariant.

```
reverse is:
    local
        prev, next:LINKABLE[G]
    do
        from
            next:=first
            first:=void
        invariant
            spliced: old model = [first ⇒^right void] □ mirror ⫩^* [next ⇒^right void]
            disjoint: [first ⇒^right void] ⫛^* [next ⇒^right void]
        until
            next = void
        loop
            prev:=first
            first:=next
            next:=next.right
            first:=set_right(prev)
        end
    require:
        reversed: model = [old model] □ mirror
end
```

We have to prove the following items:

- Initialization:
  1. $[next := first]\,\blacksquare[first := \mathbf{void}]\,\blacksquare\, spliced$
  2. $[next := first]\,\blacksquare[first := \mathbf{void}]\,\blacksquare\, disjoint$
- Invariance:
  1. $spliced \overset{*}{\wedge} disjoint \overset{*}{\Rightarrow} \overline{body}\,\blacksquare\, spliced$
  2. $spliced \overset{*}{\wedge} disjoint \overset{*}{\Rightarrow} \overline{body}\,\blacksquare\, disjoint$

Initialization (spliced)

$[next := first]\,\blacksquare[first := \mathbf{void}]\,\blacksquare[[first \overset{right}{\Rightarrow} \mathbf{void}]\,\square\, mirror \overset{*}{\#} [next \overset{right}{\Rightarrow} \mathbf{void}]]$

$=$ { dist.$\blacksquare$, $\overset{*}{\#}$, prop. 35.2 and prop.37.1 ($first \neq next \wedge first \neq \mathbf{void}$) }

$[next := first] \blacksquare [[\mathbf{void} \overset{right}{\Rightarrow} \mathbf{void}] \square mirror \overset{*}{+\!\!+} [next \overset{right}{\Rightarrow} \mathbf{void}]]$
$=$ { dist.$\blacksquare$, $\overset{*}{+\!\!+}$, prop. 35.2 and prop. 37.1 ($next \neq first$) }
$[\mathbf{void} \overset{right}{\Rightarrow} \mathbf{void}] \square mirror \overset{*}{+\!\!+} [first \overset{right}{\Rightarrow} \mathbf{void}]$
$=$ { prop. 36.5 and def.$mirror$ }
$[first \overset{right}{\Rightarrow} \mathbf{void}]$

Q.E.D.

Initialization (Disjoint)

$[next := first] \blacksquare [first := \mathbf{void}] \blacksquare [[first \overset{right}{\Rightarrow} \mathbf{void}] \square mirror \overset{*}{\pitchfork} [next \overset{right}{\Rightarrow} \mathbf{void}]]$
$=$ { dist.$\blacksquare$, $\overset{*}{\pitchfork}$, prop. 35.2 and prop.37.1 ($first \neq next$) }
$[next := first] \blacksquare [[\mathbf{void} \overset{right}{\Rightarrow} \mathbf{void}] \square mirror \overset{*}{\pitchfork} [next \overset{right}{\Rightarrow} \mathbf{void}]]$
$=$ { dist.$\blacksquare$, $\overset{*}{\pitchfork}$, prop. 35.2 and prop.37.1 ($next \neq first$) }
$[\mathbf{void} \overset{right}{\Rightarrow} \mathbf{void}] \square mirror \overset{*}{\pitchfork} [first \overset{right}{\Rightarrow} \mathbf{void}]$
$=$ { prop. 36.5 and def.$\pitchfork$ }
$true$

Q.E.D.

Invariance (spliced)

In the same way that [2], we define:

$$shift \triangleq [prev := first] \blacksquare [first := next] \blacksquare [next := next \bullet right] \tag{16}$$

and,

$$reattach \triangleq \overline{first.put\_right(previous)} \tag{17}$$

then:

If $first = \mathbf{void}$ then the proof is the same that the initialization proof, so we suppose $first \neq \mathbf{void}$:

$shift \blacksquare reattach \blacksquare [[first \overset{right}{\Rightarrow} \mathbf{void}] \square mirror \overset{*}{+\!\!+} [next \overset{right}{\Rightarrow} \mathbf{void}]]$
$=$ { def.$reattach$ }
$shift \blacksquare \overline{first.put\_right(previous)} \blacksquare [first \overset{right}{\Rightarrow} \mathbf{void}] \square mirror$
$\overset{*}{+\!\!+} [next \overset{right}{\Rightarrow} \mathbf{void}]$
$=$ { $[first \neq \mathbf{void}]$ and prop.35.5 }
$shift \blacksquare \overline{first.put\_right(prev)} \blacksquare [[first \overset{right}{\Rightarrow} first \bullet right \overset{*}{+\!\!+} first \bullet right \overset{right}{\Rightarrow} \mathbf{void}]] \square mirror$
$\overset{*}{+\!\!+} [next \overset{right}{\Rightarrow} \mathbf{void}]$

$=$ { semantics of routines, dist.$\blacksquare$, $\overset{*}{+}$, spatial separation and prop. 35.2 }

$shift\,\blacksquare[[first \overset{right}{\Rightarrow} first.right \overset{*}{+} prev \overset{right}{\Rightarrow} \mathbf{void}]\,\square\, mirror \overset{*}{+} [next \overset{right}{\Rightarrow} \mathbf{void}]]$

$=$ { def.$shift$, dist.$\blacksquare$, $\overset{*}{+}$, spatial separation and prop. 35.2 }

$[prev := first]\,\blacksquare[first := next]\,\blacksquare[[first \overset{right}{\Rightarrow} first.right \overset{*}{+} prev \overset{right}{\Rightarrow} \mathbf{void}]\,\square\, mirror \overset{*}{+} [next.right \overset{right}{\Rightarrow} \mathbf{void}]]$

$=$ { dist.$\blacksquare$, $\overset{*}{+}$, spatial separation and prop.35.6 }

$[prev := first]\,\blacksquare[[next \overset{right}{\Rightarrow} next.right \overset{*}{+} prev \overset{right}{\Rightarrow} \mathbf{void}]]\,\square\, mirror \overset{*}{+} [next.right \overset{right}{\Rightarrow} \mathbf{void}]$

$=$ { dist.$\blacksquare$, $\overset{*}{+}$, spatial separation and prop. 35.2 }

$[[next \overset{right}{\Rightarrow} next.right \overset{*}{+} first \overset{right}{\Rightarrow} \mathbf{void}]]\,\square\, mirror \overset{*}{+} [next.right \overset{right}{\Rightarrow} \mathbf{void}]$

$=$ { def.$mirror$ }

$[first \overset{right}{\Rightarrow} \mathbf{void}]\,\square\, mirror \overset{*}{+} [next \overset{right}{\Rightarrow} next.right] \overset{*}{+} [next.right \overset{right}{\Rightarrow} \mathbf{void}]$

$=$ { prop.35.5 ($next \overset{*}{\neq} \mathbf{void}$ by hyp.) }

$[first \overset{right}{\Rightarrow} \mathbf{void}]\,\square\, mirror \overset{*}{+} [next \overset{right}{\Rightarrow} \mathbf{void}]$

Q.E.D.

Invariance(disjoint):

$shift\,\blacksquare\, reattach\,\blacksquare[[first \overset{right}{\Rightarrow} \mathbf{void}] \overset{*}{\pitchfork} [next \overset{right}{\Rightarrow} \mathbf{void}]]$

$\equiv$ { def.$reattach$ }

$shift\,\blacksquare\,\overline{first.put\_right(prev)}\,\blacksquare[[first \overset{right}{\Rightarrow} \mathbf{void}] \overset{*}{\pitchfork} [next \overset{right}{\Rightarrow} \mathbf{void}]]$

$\equiv$ { $first \neq \mathbf{void}$ and prop.35.5 }

$shift\,\blacksquare\,\overline{first.put\_right(prev)}\,\blacksquare[[first \overset{f}{\Rightarrow} first.right \overset{*}{+} first.right \overset{right}{\Rightarrow} \mathbf{void}] \overset{*}{\pitchfork} [next \overset{right}{\Rightarrow} \mathbf{void}]]$

$\equiv$ { semantics of routines, dist.$\blacksquare$, $\overset{*}{\pitchfork}$, spatial separation, prop.35.7 and prop.37.2 }

$shift\,\blacksquare[[first \overset{f}{\Rightarrow} first.right \overset{*}{+} prev \overset{f}{\Rightarrow} \mathbf{void}] \overset{*}{\pitchfork} [next \overset{right}{\Rightarrow} \mathbf{void}]]$

$\equiv$ { def.$shift$, dist.$\blacksquare$, $\overset{*}{+}$, spatial separation and prop.35.2 }

$[prev := first]\,\blacksquare[first := next]\,\blacksquare[[first \overset{f}{\Rightarrow} first.right \overset{*}{+} prev \overset{f}{\Rightarrow} \mathbf{void}] \overset{*}{\pitchfork} [next.right \overset{right}{\Rightarrow} \mathbf{void}]]$

$\equiv$ { dist.$\blacksquare$, $\overset{*}{+}$, spatial separation and prop.35.5 }

$[prev := first]\,\blacksquare[[next \overset{f}{\Rightarrow} next.right \overset{*}{+} prev \overset{f}{\Rightarrow} \mathbf{void}] \overset{*}{\pitchfork} [next.right \overset{right}{\Rightarrow} \mathbf{void}]]$

$\equiv$ { dist.$\blacksquare$, $\overset{*}{+}$, spatial separation and prop.35.2 }

$[next \overset{f}{\Rightarrow} next.right \overset{*}{+} first \overset{f}{\Rightarrow} \mathbf{void}] \overset{*}{\pitchfork} [next.right \overset{right}{\Rightarrow} \mathbf{void}]$

$\equiv$ { properties of $\pitchfork$ }

$[[next \overset{f}{\Rightarrow} next \bullet right] \overset{*}{\pitchfork} [next \bullet right \overset{right}{\Rightarrow} \mathbf{void}]]$
$\overset{*}{\wedge}$
$[first \overset{f}{\Rightarrow} \mathbf{void}] \overset{*}{\pitchfork} [next \bullet right \overset{right}{\Rightarrow} \mathbf{void}]$
$\equiv$ { prop.38 and hyp. }
$true \overset{*}{\wedge} true$
$\equiv$ { logic }
$true$

Q.E.D.

11 A new model for LINKED_LIST

In this section we propose a new model for `LINKED\_LIST` structures, the benefits of this new approach are several in comparation with the classic model. The main idea is to use the fixed-point theory to find a suitable model, which ensures the well-defineness, and additionally allows us to reasons easily about linked structures. First, consider the following function defined by induction on $\mathbb{N}$:

$$0.p \overset{f}{\Rightarrow} q \overset{\triangle}{=} \overset{*}{\langle\rangle}$$
$$N+1.p \overset{f}{\Rightarrow} q \overset{\triangle}{=} \mathsf{function}\ o \mid \mathsf{function}\ s \mid \mathsf{if}\ [[p \overset{*}{=} q] \overset{*}{\vee} [p \overset{*}{=} \mathbf{void}]](o)(s)\ then\ \langle\rangle \\ \mathsf{else}\ \langle p(o)(s)\rangle \# [p \bullet f \overset{f}{\Rightarrow} q](o)(s)$$

Where $p, q : Objects \nrightarrow States \nrightarrow Objects$ and $f : Names$. In other words, this function returns the path between p and q which has at most N objects. Note that this function is total, that is it is always well-defined. Using this operator new we can consider the following function:

$$\Phi_{p,q,o,s} \overset{\triangle}{=} \mathsf{function}\ n \mid diff[n.[p \overset{f}{\Rightarrow} q](o)(s)] + 1 \tag{18}$$

Where $diff$ is a function which counts the number of differents elements in the list. It is easy to see that $\Phi_{p,q,o,s}$ is monotonic:

$$n \leq m \Rightarrow \Phi_{p,q,o,s}(n) \leq \Phi_{p,q,o,s}(m)$$

Since the number of different objects is finite, this function is also continuos, in the sense of CPO theory (see []). And therefore, it has a least fixed point. Actually, the least fixed point of this function can be defined (calculated) as follows:

$$fix(\phi_{p,q,o,s}) = Max_{n \in \mathbb{N}} \Phi^n_{p,q,o,s}(0)$$

That is, it can be calculated using approximations starting at the bottom element (0). In other words, we use the Knaster-Tarski theorem. Using fix we define:

$$[\mu.p \overset{f}{\Rightarrow} q] \overset{\triangle}{=} \mathsf{function} o \mid \mathsf{function}\ s \mid [[fix(\Phi_{p,q,o,s}) - 1].p \overset{f}{\Rightarrow} q](o)(s) \tag{19}$$

And this is our new model. Note that the $+/-1$ are neccesary to obtain the correct model using the Knaster-Tarski theorem.

Intuitively, $[\mu.p \overset{f}{\Rightarrow}]$ evaluates in o and s returns the longest list between p and q which does not have repeated elements. Two important theorems about this construction is:

Theorem 4 *Let $p, q, x, y : Objects \nrightarrow States \nrightarrow Objects$, $t, h : Objects \nrightarrow States \nrightarrow States$ and $f : Names$ then:*

$$\forall n : \mathbb{N} \bullet t\,\blacksquare[n.p \overset{f}{\Rightarrow} q] = h\,\blacksquare[n.x \overset{f}{\Rightarrow} y] \overset{*}{\Rightarrow} [t\,\blacksquare[\mu.p \overset{f}{\Rightarrow} q] = h\,\blacksquare[\mu.p \overset{f}{\Rightarrow} q]]$$

Proof.*By the definition of the μ construction our theorem is reduced to prove:*

$$[Max_{n\in\mathbb{N}}\Phi_{p,q,o,s}(0)] = [Max_{n\in\mathbb{N}}\Phi_{p,q,o,s}(0)]$$

But, using the hypothesis and the functionality of Max we have:

$$[t\,\blacksquare[\mu.p \overset{f}{\Rightarrow} q] = h\,\blacksquare[\mu.p \overset{f}{\Rightarrow} q]]$$

Q.E.D.

Note the relevance of this theorem: we can use induction to prove properties on the fixed-point model. The next theorem expreses the distributivity of $\blacksquare$ with respect to $\mu.\overset{f}{\Rightarrow}$. Let us see:

Theorem 5 *Let $h : Objects \nrightarrow States \nrightarrow States$ and $p, q : Objects \nrightarrow States \nrightarrow Objects$ then:*

$$h\,\blacksquare[\mu.p \overset{f}{\Rightarrow} q] \overset{*}{=} [\mu.[h\,\blacksquare p] \overset{f}{\Rightarrow} [h\,\blacksquare q]]$$

Proof.*Since theorem 4 we can use induction to prove this one.*
Base Case) N=0:

$h\,\blacksquare[0.p \overset{f}{\Rightarrow} q]$
$=$ { *def. of* $[0.p \overset{f}{\Rightarrow} q]$ }
$\overset{*}{\langle\rangle}$
$=$ { *def. of* $[0.p \overset{f}{\Rightarrow} q]$ }
$[0.[h\,\blacksquare p] \overset{f}{\Rightarrow} [h\,\blacksquare q]]$
I.H. Suppose that the property holds for an arbitrary N.
I.C. The case when $p \overset{*}{=} q \vee p \overset{*}{=}$ **void** *is straighforward. Then consider the other case:*

$\langle p(o)(h(o)(s))\rangle +\!\!+ [N.[p \bullet f] \overset{f}{\Rightarrow} q](o)(h(o)(s))$
$=$ { *def. of* $\blacksquare$ }
$\langle [h\,\blacksquare p](o)(s) +\!\!+ [h\,\blacksquare[N.p \bullet f \overset{f}{\Rightarrow} q]](o)(s)$
$=$ { *I.H.* }
$\langle [h\,\blacksquare p](o)(s) +\!\!+ [N.[h\,\blacksquare p] \bullet f \overset{f}{\Rightarrow} [h\,\blacksquare q]](o)(s)$
$=$ { *Def. of* $\overset{f}{\Rightarrow}$ *and case 2 hyp.* }
$[N+1.[h\,\blacksquare p] \overset{f}{\Rightarrow} [h\,\blacksquare q]]$
Q.E.D.

Using these two theorems a set of interesting properties can be proved:

Theorem 6 *Let f, h, g different elements of Names and p, q, x, y : Objects $\nrightarrow$ States $\nrightarrow$ Objects then:*

1. $[h := q]\, \blacksquare[\mu.\overline{g} \overset{f}{\Rightarrow} \overline{h}] \overset{*}{=} [\mu.\overline{g} \overset{f}{\Rightarrow} q]$
2. $[h := p]\, \blacksquare[\mu.\overline{h} \overset{f}{\Rightarrow} \overline{g}] \overset{*}{=} [\mu.p \overset{f}{\Rightarrow} \overline{g}]$
3. $[p \overset{*}{\neq} q] \overset{*}{\Rightarrow} [\mu.p \overset{f}{\Rightarrow} q \overset{*}{=} \mu.p \overset{f}{\Rightarrow} [p \bullet \overline{f}] \overset{*}{+\!\!+} \mu.[p \bullet \overline{f}] \overset{f}{\Rightarrow} q]$
4. $[p \overset{*}{\neq} \textbf{void}] \overset{*}{\Rightarrow} [\![p \bullet [f := \underline{h}]]\!]\, \blacksquare[\mu.[p \bullet \overline{f}] \overset{f}{\Rightarrow} \overline{g}] \overset{*}{=} [\mu.h \overset{f}{\Rightarrow} \overline{g}]$
5. $[x \overset{*}{\neq} \mathsf{void}] \overset{*}{\wedge} [x \overset{*}{\notin} [\mu.p \overset{f}{\Rightarrow} q]] \overset{*}{\Rightarrow} [[x \bullet [f := y]]\, \blacksquare[\mu.p \overset{f}{\Rightarrow} q] \overset{*}{=} [\mu.p \overset{f}{\Rightarrow} q]]$
6. $[h := q]\, \blacksquare[\mu.\overline{h} \overset{f}{\Rightarrow} [\overline{h} \bullet \overline{f}]] \overset{*}{=} [\mu.q \overset{f}{\Rightarrow} [q \bullet \overline{f}]]$

Proof.
1, 2, 4 and 6 are straighforward applying theorem 5, and properties of :=. The proof of property 5 is similar to the proof of the theorem 3. Finaly, the property 3 can be proved using the following property:

$$[N + 1.p \overset{f}{\Rightarrow} q] \overset{*}{=} [1.p \overset{f}{\Rightarrow} p \bullet f] \overset{*}{+\!\!+} [N.p \bullet f \overset{f}{\Rightarrow} q]$$

which proof is straighforward by induction and definition of $N.\overset{f}{\Rightarrow}$, from here and recalling the definition of least fixed point we have the proof.
Q.E.D.

12 Conclusions and further work

In this article a formal semantics for an important subset of the Eiffel language is fully defined. The main characteristics of this definition (shared with [2]) are its simplicity and the natural way in which pointer operations can be expressed. Furthermore, the ubiquitous use of function composition instead of application solves many problems concerning undefined values. We tried to correct some mistakes of the original presentation and to clarify some points. This is the reason to present many times complete detailed proofs. Moreover, if the formalism is aimed to be used in practice, it may have to deal with complete proofs. We can conclude that this is indeed feasible but there is an explosion of properties that need to be proven in order to verify even simple programs. We hope this process can converge (at least for a given class) when enough programs are verified and hence most of the relevant properties are already proven. The distributivity of functional composition with respect to many model operations, seems to be essential to modularize proofs and hence it may prove to be an important tool to use in mechanizing them.

We showed that some other approaches based on Hoare logic to verify pointer programs can be embedded easily in this framework. This can be important to translate proofs made in other contexts into this one.

References

[1] B.Meyer. *Object-Oriented Software Construction*, Second Edition, Prentice Hall, 1997.

[2] B.Meyer. *Towards Practical Proofs of Class Correctness.* ZB 2003: Formal Specification and Development in Z and B, Proceedings of 3rd International Conference, Turku, Finland, June 2003. Lectures Notes in Computer Science 2651, Springer Verlag, 2003.

[3] B.Meyer. *A Framework for Proving Contract-Equipped Classes.* Abstract State Machines 2003 - Advances in Theory and Applications, Proc. 10th International Workshop, Toarmina, Italy, March 3-7.

[4] B.Meyer. *Proving Pointer Program Properties.* Draft version available at http://www.inf.ethz.ch/personal/meyer/ongoing/references/

[5] R.Bornat. *Proving Pointer Programs in Hoare Logic.* In *Mathematics of Program Construction*, Springer-Verlag, 2000.

[6] Richard Bird. *Introduction to Functional Programming using Haskell*, Second Edition, Prentice Hall Series in Computer Science, 1998.

[7] C.A.R.Hoare. *An Axiomatic Basis for Computer Programming.* Communications of the ACM, 12(10):576-580 and 583. October 1969.

[8] D.A.Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc., 1986.

[9] B.Meyer. *Eiffel, The Language.* Prentice Hall, Englewood Cliffs, 1992.

[10] R.J.Back and J.Wright. *Refinement Calculus: A Systematic Introduction.* Springer. 1997.

[11] R.J.Back, Xiaocong Fan and Viorel Preoteasa. *Reasoning about Pointers in Refinement Calculus*. Turku Centre for Computer Science, TUCS Technical Report No 543.

[12] R.Paige, J.Ostroff and P.Brooke. *Formalising Eiffel Reference and Expanded Types in PVS*. In Proc. International Workshop on Aliasing, Confinment, and Ownership in Object-Oriented Programming, Darmstadt, Gernamy, Lujy 2003.

[13] Fardah Mehta and Tobias Nipkow. *Proving Pointers Program in High Order Logic*. CADE 2003.