Towards Practical Class Proofs

Bertrand Meyer

ETH Zurich & Eiffel Software

PURPOSE

Proving properties of O-O programs and their run-time structures, including pointers.

WHY DO THIS?

Trusted Components

RELATED WORK

Framework for proving classes

- **Notion of model**
- **Taking advantage of inheritance**
- **Factoring out parts of proofs**

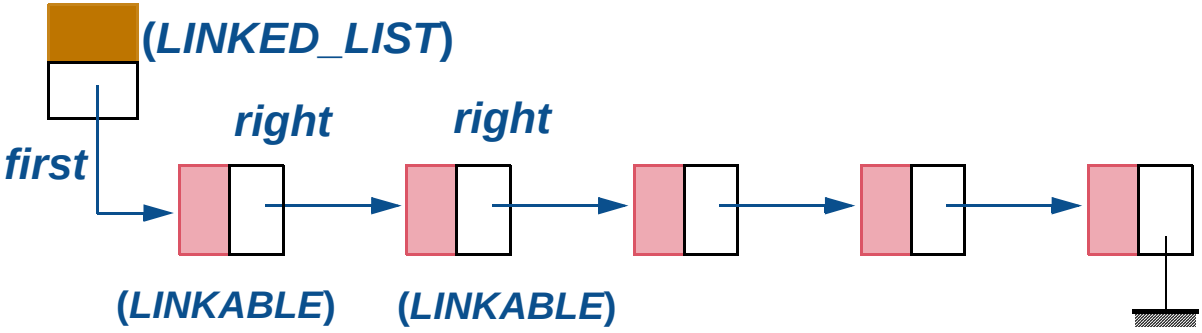
Work on pointers

(See October 2001 presentation, Dartmouth)

GUIDELINES

- Start from the object structure
- Model pointers
- Focus on *Current*
- Use high-level functions and operators
- Use relations, partial functions
- Use models

LINKED LISTS IN EIFFELBASE

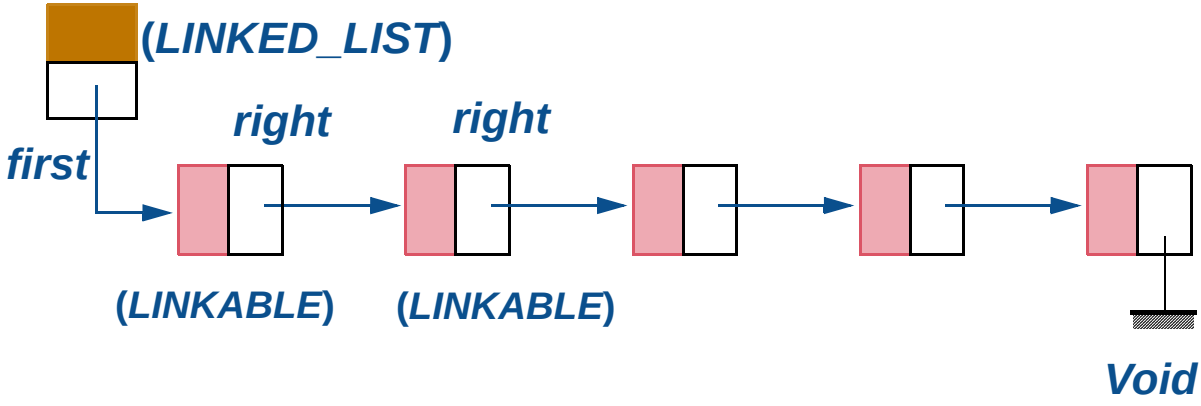


THINGS TO PROVE

Starting from *first* and following *right* links:

- **No element encountered twice**
- **Eventually reaches a *Void***

An insertion will keep the previous elements, with their previous index or (right of the insertion) the previous index plus one etc.



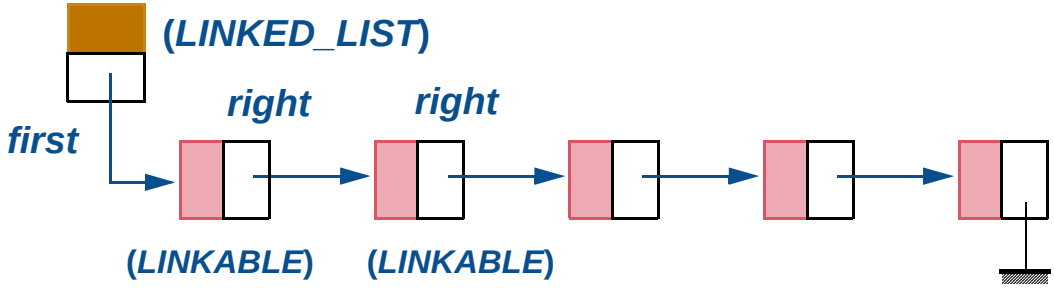
PROGRAMMING THE LINKED LIST

class *LINKED_LIST* [*G*] **feature**

first: *LINKABLE* [*G*]

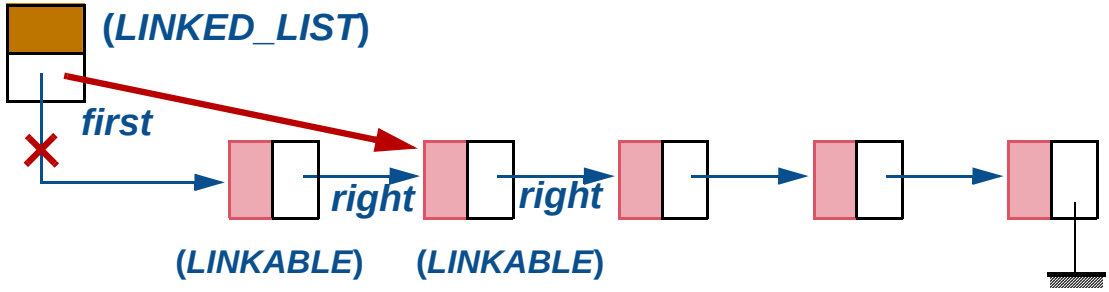
... Routines (see below) ...

end



REMOVING AN ELEMENT

```
remove_front is
        -- Remove first element of list.
    require
        not_empty: first /= Void
    do
        first := first.right
    ensure
        ...
    end
```

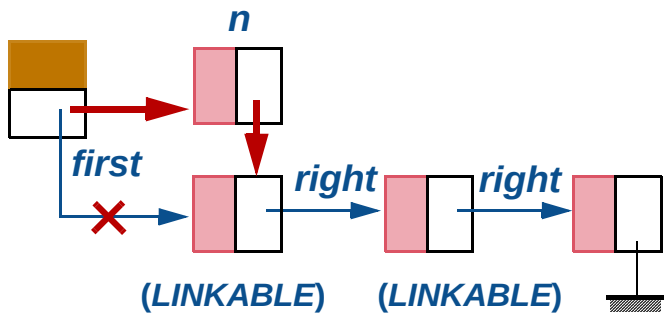


INSERTING AN ELEMENT

```
put_front is
        -- Add element to beginning of list
    local
        n: LINKABLE [G]
    do
        create n
        n.put_right (first)
        first := n
    ensure
        ...
    end
```

Java, C++:

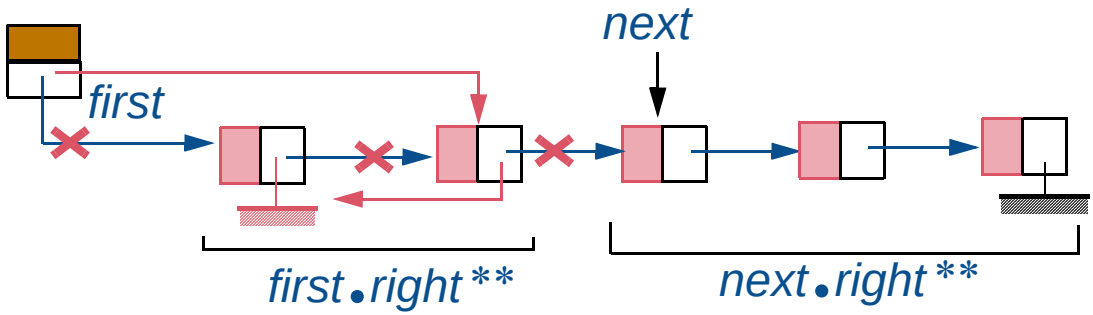
```
n.right = first
```



In class *LINKABLE*:

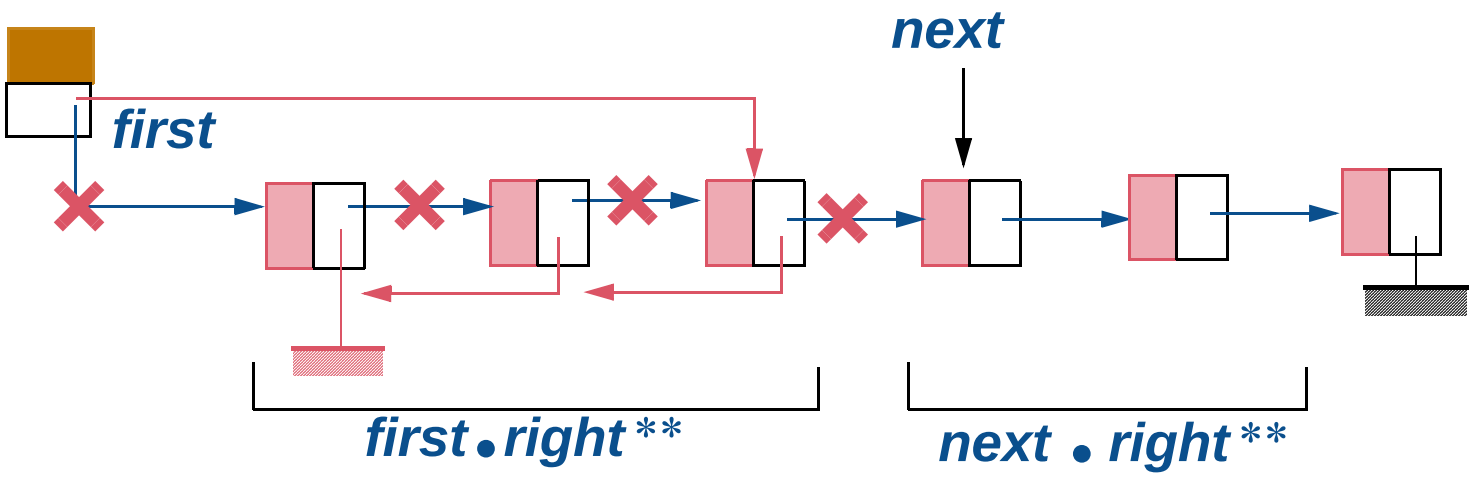
```
put_right (other: LINKABLE [G]) is
        -- Chain current object to other
    do
        right := other
    end
```

REVERSING THE LIST




```
reverse is
    local
        previous, next: LINKABLE [G]
    do
        from next := first invariant
            spliced: old model = first.right** ◻ mirror + next.right**
                until next = Void loop
            [previous, first, next] := [first, next, next.right]
            first.put_right (previous)
        end
    ensure
        reversed: model = old model.mirror
    end
```

INTERMEDIATE STATE

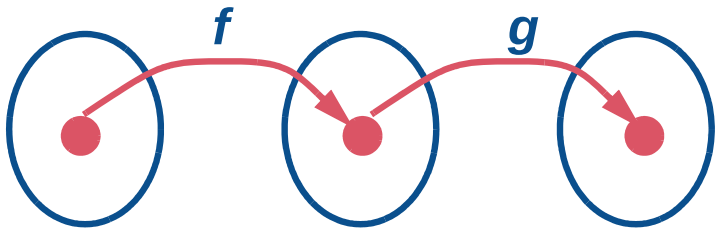


Invariant:

old *model = first . right ** □ mirror + next . right **

NOTATION FOR BASIC COMPOSITION

Apply *f* then *g*:



This will be written:

$$f \square g$$

Defined by

$$[f \square g]\ (a) \ = \ g\ (f\ (a))$$

Similar to usual $g \circ f$ but applied in reverse order.

Works for total functions, partial functions, relations.

RELYING ON PARTIAL FUNCTIONS

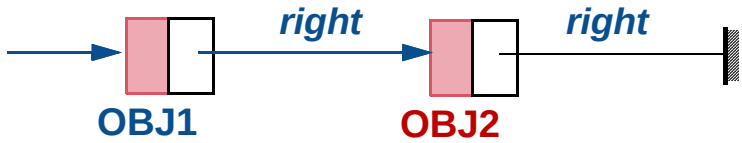
Modeling power:

- Attributes are functions on objects; defined only on the corresponding classes, e.g.

$$salary: \; Objects \rightarrow \mathbb{N}$$

$$\text{domain } (salary) \subseteq Employees$$

- Void references: simply a function that's not defined for a certain element



$$OBJ2 \notin \text{domain } (right)$$

- Avoid application $f\ (a)$: use composition, image...

BRACKETS AND PARENTHESES

Function application: parentheses

$f\ (a)$

Grouping: brackets only

$f \circ [g \circ h]\ (a) = [f \circ g] \circ h\ (a)$

(Associativity of composition)

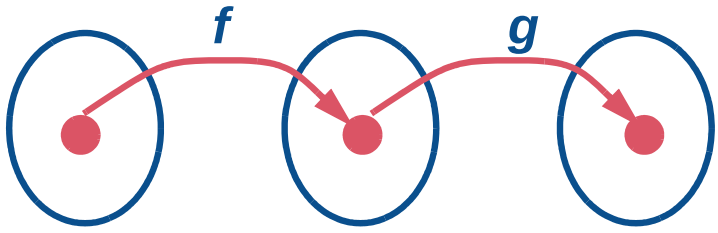
LAMBDA-LIKE NOTATION

function *x* **|** *expression_that_may_involve_x*

Defines a function

COMPOSITION 1: BASIC

Apply *f* then *g*:



This will be written:

$$f \circ g$$

COMPOSITION 2: CURRIED ON STATE

Consider functions

$$f: A \xrightarrow{\sim} [States \xrightarrow{\sim} B]$$

$$g: B \xrightarrow{\sim} [States \xrightarrow{\sim} C]$$

Then $f \square g$ is meaningless, but can compose f and g applied to a given state s.

$f \bullet g$ is that composition, with signature

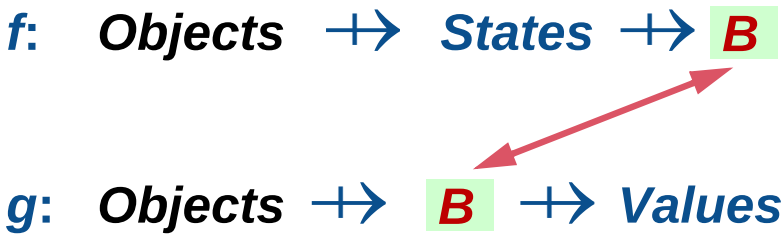
$$A \xrightarrow{\sim} [States \xrightarrow{\sim} C]$$

and value

function x | [function s | [g ([f (x)] (s))] (s)]

COMPOSITION 3: RIGHTMOST (MORE CURRY SOUP)

Consider functions



Then *f* ◻ *g* is meaningless, but the following is defined:

function obj | *f* (*obj*) ◻ *g* (*obj*)

This will be written

f* ■ *g

OTHER "RIGHTMOST" OPERATORS

For *f*: *Objects* $\rightarrow\!\!\!\!\!+$ *B* $\rightarrow\!\!\!\!\!+$ *C*:

f^{-1*}

denotes:

function obj | $[f\ (obj)]^{-1}$

Also rightmost implication $\overset{*}{\Rightarrow}$, rightmost conjunction $\overset{*}{\wedge}$.

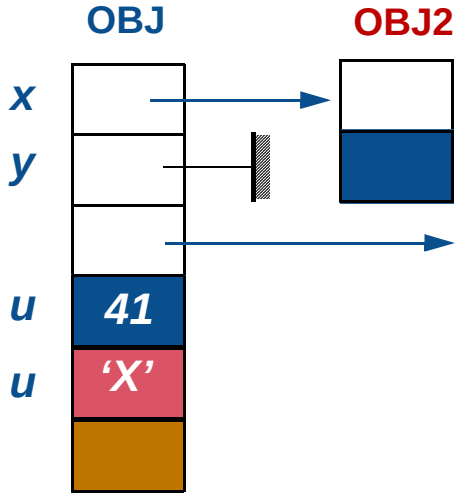
STATE AND STATE CHANGE

What's a state of object-oriented program execution?

- The set *Objects*, a subset of $\mathbb{N}$
  (objects identified by addresses)
- Zero or more functions

  $x, y, \ldots$: *Objects* $\rightarrow$ *Objects*

  $u, v, \ldots$: *Objects* $\rightarrow$ *Values*



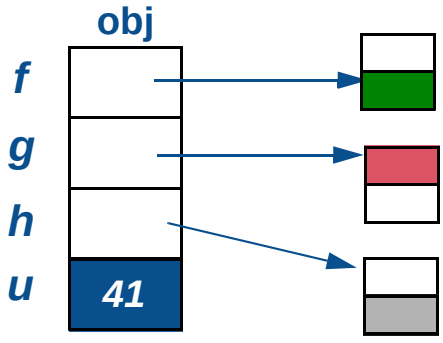
What's a state change?

- A change to the set *Objects* or to one of the functions

BASIC OPERATION ON FUNCTIONS

Function substitution:

$$f := g$$



Denotes function in

$$Objects \rightarrow States \rightarrow States$$

where for any *obj*, [*f* (*obj*)] (*s*) is state *s'* that differs from *s* only by

$$[g\ (obj)]\ (s') = [g\ (obj)]\ (s)$$

(or undefined if *obj* not in domain of *g*).

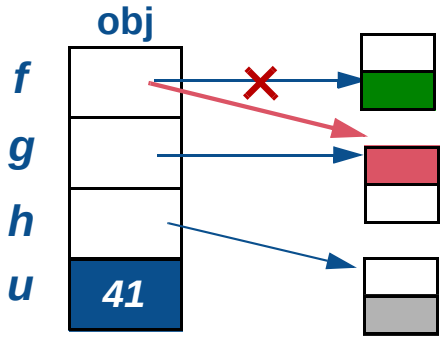
Characteristic properties:

$$[f := g] \blacksquare f = g$$
$$[f := g] \blacksquare h = h \qquad \text{-- For } h \text{ other than } f$$

BASIC OPERATION ON FUNCTIONS

Function substitution:

$$f := g$$



Denotes function in

$$Objects \pfun States \pfun States$$

where for any *obj*, [*f* (*obj*)] (*s*) is state *s'* that differs from *s* only by

$$[g\ (obj)]\ (s') = [g\ (obj)]\ (s)$$

(or undefined if *obj* not in domain of *g*).

Characteristic properties:

$$[f := g] \bullet f = g$$
$$[f := g] \bullet h = h \qquad \text{-- For } h \text{ other than } f$$

SEMANTIC RULES: BASICS

Construct		Denotation	Signature
$\overline{f}$	=	f	Objects → States → Values
$\overline{f := g}$	=	$\overline{f} := \overline{g}$	Objects → States → States
$\overline{f1, f2 := g1, g2}$	=	$\overline{f1}, \overline{f2} := \overline{g1}, \overline{g2}$	Objects → States → States
$\overline{i ; j}$	=	$\overline{i} \blacksquare \overline{j}$	Objects → States → States
$\overline{x} \bullet \overline{f}$	=	$\overline{x} \bullet \overline{f}$	Objects → States → Objects

RULES: ROUTINES AND CALLS

$\overline{r}$	=	function a | $\overline{body_r}$	$Values \to Objects \to States \to States$
$\overline{r\ (u)}$	=	$\overline{r}\ (\overline{u})$	$Objects \to States \to States$ -- For procedure $Objects \to States \to Values$ -- For function
$\overline{x \bullet r\,(u)}$	=	$\overline{x} \bullet \overline{r}\ (\overline{u})$	Same
$\overline{\S\ a}$	=	$\overline{\S}\ \overline{a}$	$Objects \to States \to Values$
$\overline{a\ \S\ b}$	=	$\overline{a}\ \overline{\S}\ \overline{b}$	$Objects \to States \to Values$

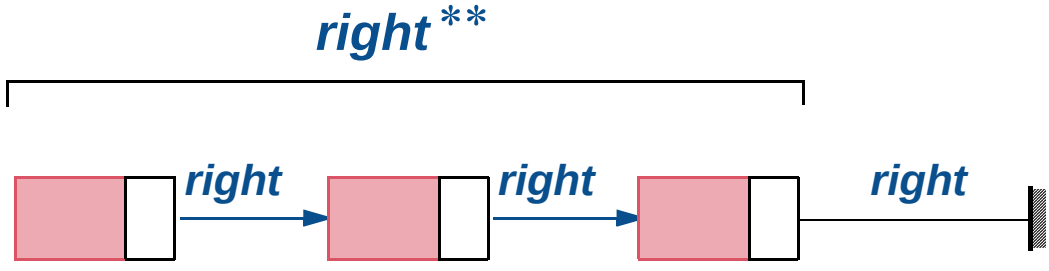
RULES: ASSERTIONS

In a routine *r*:

$\overline{\text{ensure } Q}$	$= \overline{r} \,.\, \overline{Q}$	$\mathit{Values} \to \mathit{Objects} \to \mathit{States} \to \mathbb{B}$
$\overline{\text{old } Q}$	$= \overline{r}^{-1*} \,.\, \overline{f}$	Same as *f*
$\overline{\text{ensure } f = \text{old } g}$	$= [\, \overline{r} \,.\, \overline{f} = \overline{g} \,]$	$\mathit{Objects} \to \mathit{States} \to \mathbb{B}$
$\overline{\text{ensure } Q}$	$= \overline{r} \,.\, \overline{Q}$	$\mathit{Values} \to \mathit{Objects} \to \mathit{States} \to \mathbb{B}$
$[\overline{pre} \overset{*}{\wedge} \overline{inv} \overset{*}{\Rightarrow} \overline{r} \,.\, [\overline{post} \overset{*}{\wedge} \overline{inv}]]$		$\mathbb{B}$

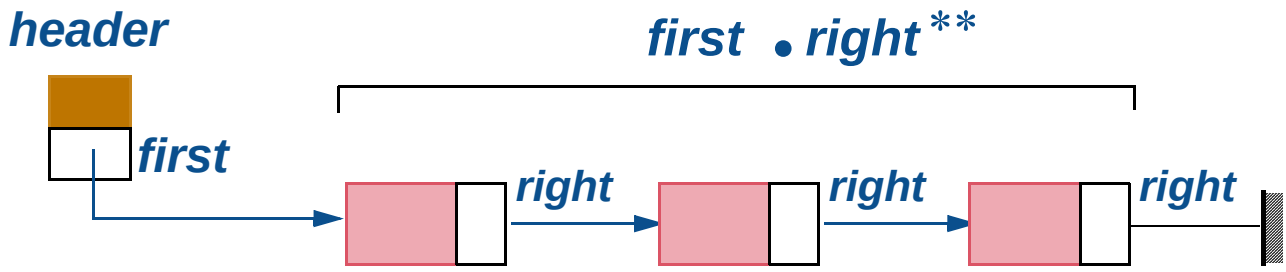
MODELING LISTS

Sequence closure: *right***



MODELING LISTS

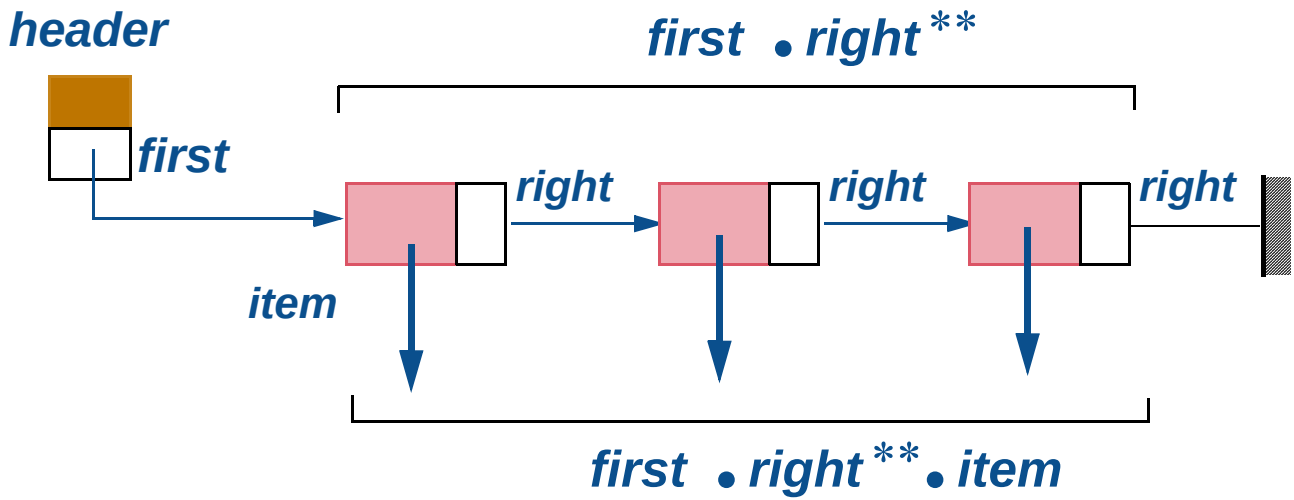
Composition and sequence closure:



MODELING LISTS

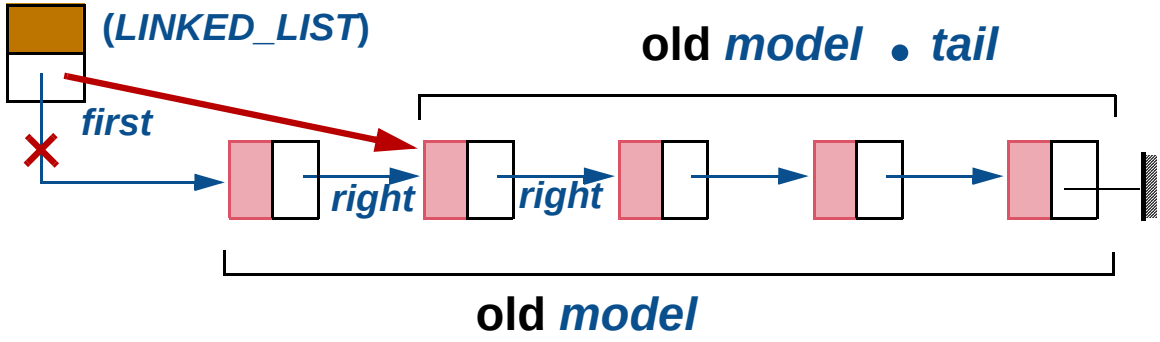
Composition and sequence closure:

Class invariant: $model = first \bullet right^{**} \bullet item$



REMOVING AN ELEMENT

```
remove_front is
        -- Remove first element of list.
    require
        not_empty: first /= Void
    do
        first := first.right
    ensure
        model = old model . tail
    end
```



PROVING THE REMOVAL ROUTINE

We have to prove

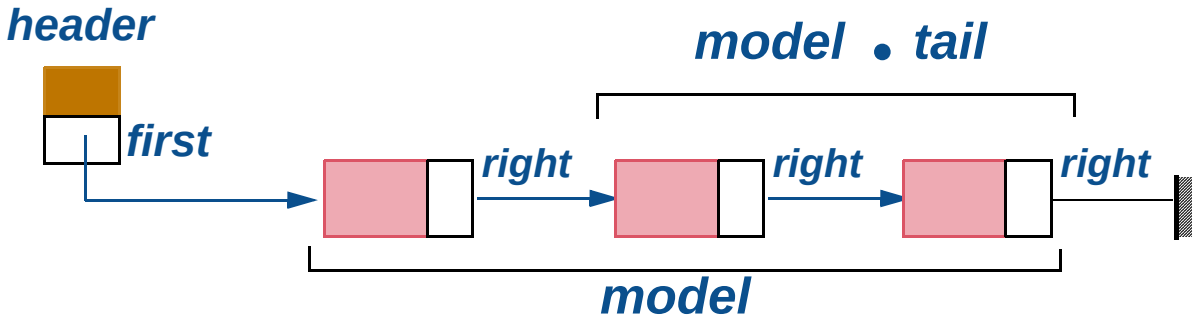
$$\overline{remove\_front} \blacksquare model = model \bullet tail$$

From the semantic rules

$$\overline{remove\_front} = [first := first \bullet right]$$

Model property:

$$[first := first \bullet right] \blacksquare model = model \bullet tail$$



BASIC OPERATION ON FUNCTIONS

Function substitution:

$$f := g$$

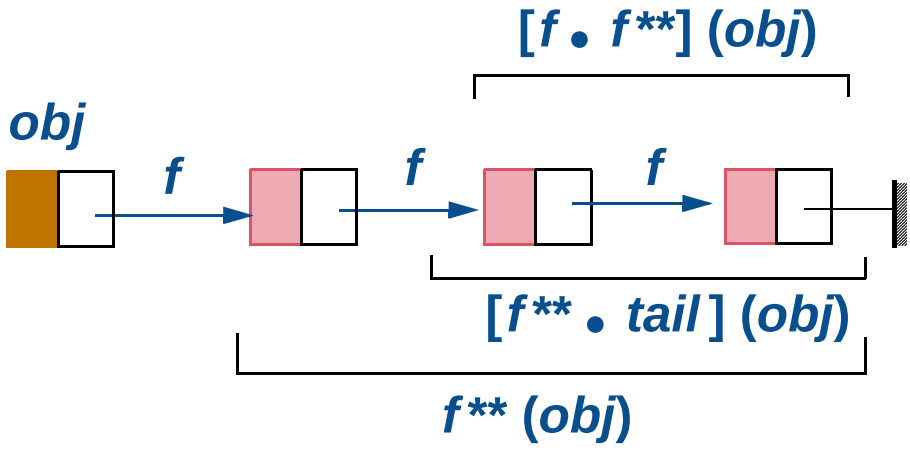
Denotes function in *Objects* $\rightarrow$ *States* $\rightarrow$ *States* such that, for any *obj*, *f* (*obj*) in resulting state is *g* (*obj*) (or undefined if *obj* not in domain of *g*).

Characteristic properties:

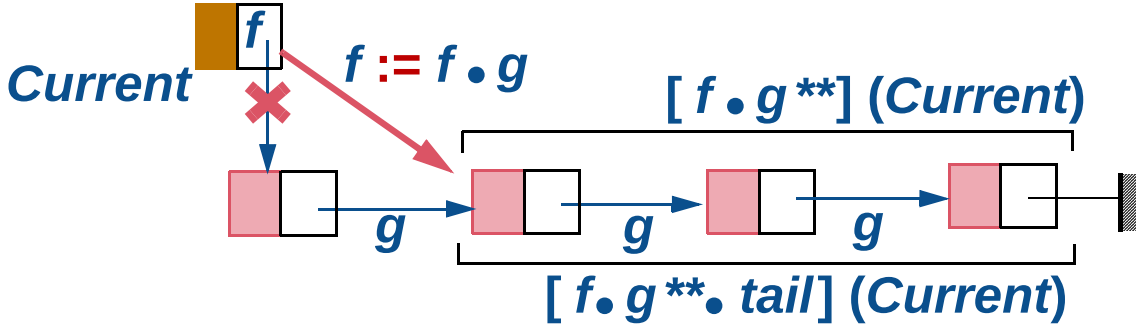
$$[f := g] \blacksquare f = g$$
$$[f := g] \blacksquare h = h \qquad \text{-- For } h \text{ other than } f$$

PROPERTIES OF TAIL AND SUBSTITUTION

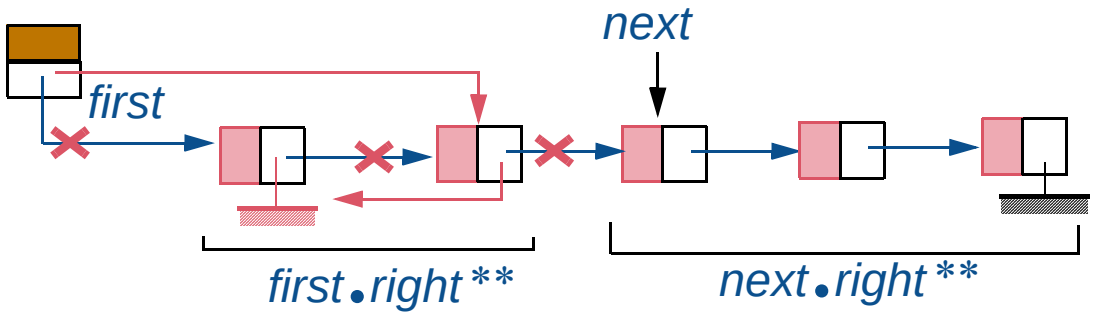
$f \bullet f^{**} = f^{**} \bullet tail$



$[f := f \bullet g] \blacksquare [f \bullet g^{**}] = f \bullet g^{**} \bullet tail$



REVERSING THE LIST

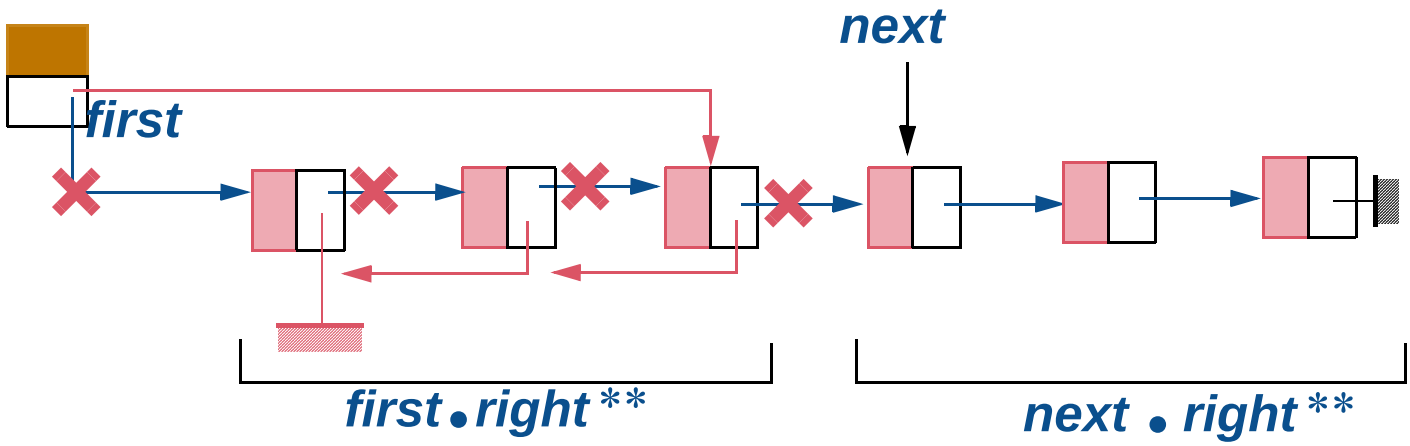



```
reverse is
    local
        previous, next: LINKABLE [G]
    do
        from next := first invariant
            spliced: old model = first.right** ◻ mirror + next.right**
                until next = Void loop
            [previous, first, next] := [first, next, next.right]
            first.put_right (previous)
        end
    ensure
        reversed: model = old model.mirror
    end
```

PROVING THE REVERSAL ROUTINE

Loop invariant:

old *model = first . right** . mirror + next . right**



PROOF SKETCH

We have to prove that

$$\overline{spliced} \quad \stackrel{*}{\Rightarrow} \quad \overline{Body} \blacksquare \overline{spliced}$$

where *Body* is the body of the loop:

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

We compute $\overline{Body} \blacksquare spliced$. This is

$$[\overline{Shift} \blacksquare \overline{Reattach}] \blacksquare spliced$$

Associativity applies: first compute $\overline{Reattach} \blacksquare spliced$.

CONTINUED

Reattach is *first* ▪ *put_right* (*previous*).

Procedure *put_right* (*x*), in class *LINKABLE*, performs the assignment *right* := *x*, so its semantics is

put_right = function *a* | *right* := *a*

giving:

Reattach = *first* ▪ [*right* := *previous*]

hence

Reattach ▪ *spliced* =
[old *model* =
[*first* ▪ [*right* := *previous*]] ▪
[*first* .*right*** ▫ *mirror* + *next* .*right***]]

CONTINUED

Value is:

$$\overline{Reattach} \blacksquare spliced = [\text{old } model = [<first> + previous.right^{**}] \square mirror + next.right^{**}]$$

We are computing $\overline{Shift} \blacksquare \overline{Reattach} \blacksquare spliced$**. Applying multiple assignment axiom [S7]:**

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

Yielding

$$\text{old } model = [first.right^{**}] \square mirror + <next> + next.right.right^{**}]$$

SEMANTIC RULES: BASICS

Construct		Denotation	Signature
$\overline{f}$	=	f	$Objects \to States \to Values$
$\overline{f := g}$	=	$\overline{f} := \overline{g}$	$Objects \to States \to States$
$\overline{f1, f2 := g1, g2}$	=	$\overline{f1}, \overline{f2} := \overline{g1}, \overline{g2}$	$Objects \to States \to States$
$\overline{i ; j}$	=	$\overline{i} \blacksquare \overline{j}$	$Objects \to States \to States$
$\overline{x} \bullet \overline{f}$	=	$\overline{x} \bullet \overline{f}$	$Objects \to States \to Objects$

RULES: ROUTINES AND CALLS

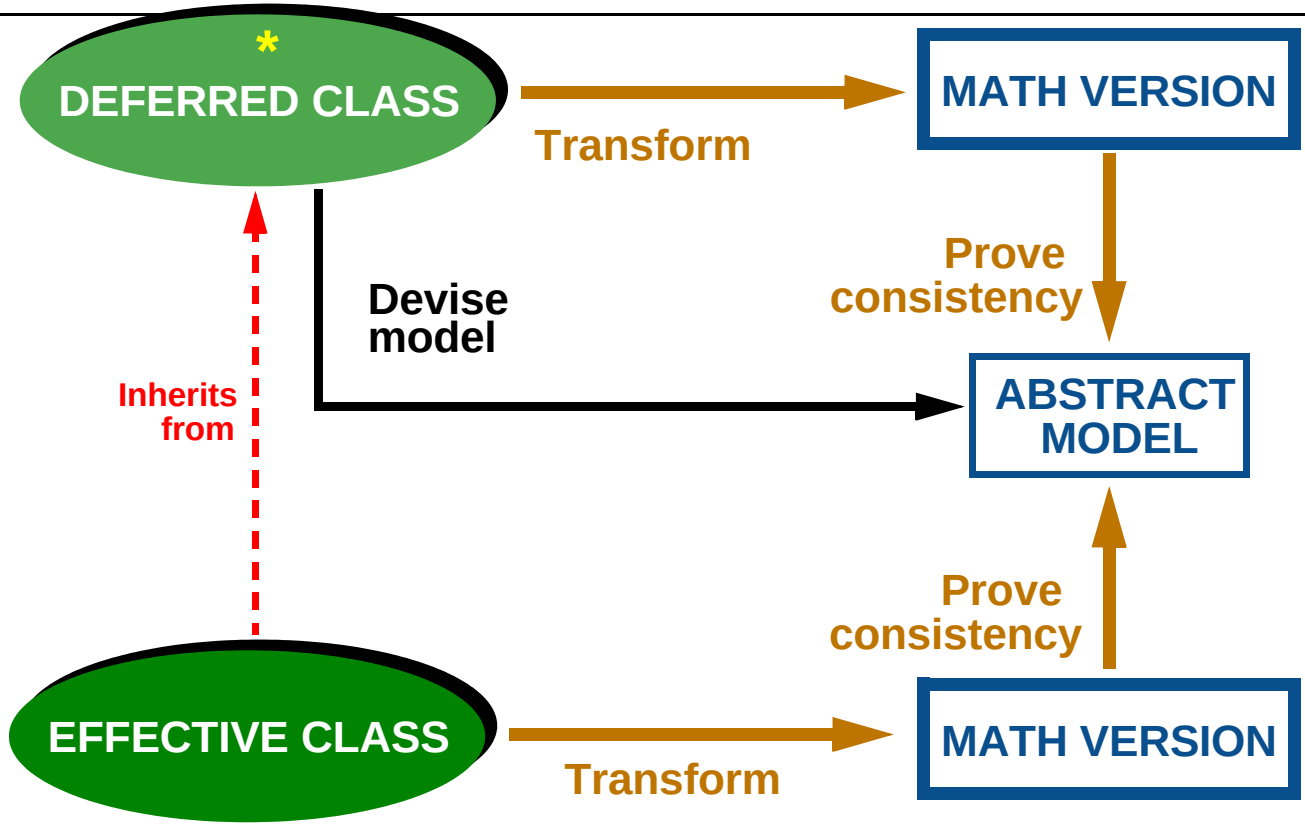
$\overline{r}$ =	function a | $\overline{body_r}$	$Values \to Objects \to States \to States$
$\overline{r\ (u)}$ =	$\overline{r}\ (\overline{u})$	$Objects \to States \to States$ -- For procedure $Objects \to States \to Values$ -- For function
$\overline{x \bullet r\,(u)}$ =	$\overline{x} \bullet \overline{r}\ (\overline{u})$	Same
$\overline{\S\ a}$ =	$\overline{\S}\ \overline{a}$	$Objects \to States \to Values$
$\overline{a\ \S\ b}$ =	$\overline{a}\ \overline{\S}\ \overline{b}$	$Objects \to States \to Values$

RULES: ASSERTIONS

In a routine *r*:

$\overline{\text{ensure } Q}$	$= \overline{r} \,.\, \overline{Q}$	$\mathit{Values} \to \mathit{Objects} \to \mathit{States} \to \mathbb{B}$
$\overline{\text{old } Q}$	$= \overline{r}^{-1*} \,.\, \overline{f}$	Same as *f*
$\overline{\text{ensure } f = \text{old } g}$	$= [\, \overline{r} \,.\, \overline{f} = \overline{g} \,]$	$\mathit{Objects} \to \mathit{States} \to \mathbb{B}$
$\overline{\text{ensure } Q}$	$= \overline{r} \,.\, \overline{Q}$	$\mathit{Values} \to \mathit{Objects} \to \mathit{States} \to \mathbb{B}$
$[\overline{pre} \overset{*}{\wedge} \overline{inv} \overset{*}{\Rightarrow} \overline{r} \,.\, [\overline{post} \overset{*}{\wedge} \overline{inv}]]$		$\mathbb{B}$

TAKING ADVANTAGE OF INHERITANCE


*
DEFERRED CLASS
Transform
MATH VERSION
Prove
consistency
Devise
model
Inherits
from
ABSTRACT
MODEL
Prove
consistency
EFFECTIVE CLASS
Transform
MATH VERSION

PROCESS

- Devise a model.
- Build a static theory. .
- Extend the contracts.
- Translate the class to mathematical form.
- Perform the proofs.

- Prove that the abstract assertions imply the model assertions in the deferred class.
- Prove the consistency of the concrete model against the abstract model in the effective class.vb

ISSUES

Getting all the details right

Circumscribing the effect of operations of a class

Developing the models

Developing the Eiffel to math translation

Feeding this into a proof engine

Integrating with the other part of the O-O method: inheritance, polymorphism, dynamic binding