Marktoberdorf 2004

Towards Trusted Components

Bertrand Meyer

ETH, Zürich & Eiffel Software, California

Lesson 2:
Contracts &
the overall pointer structure

Contracts in Eiffel

At the routine level:

- Preconditions
- Postconditions

At the class level:

- Class invariant

Other assertion constructs:

- "**check**" instruction
- Loop invariant and variant

(From lesson 1) Component Quality Model

A: Acceptance

B: Behavior

- B.1 Examples
- B.2 Usage documentation
- B.3 Preconditioned
- B.4 Some postconditions
- B.5 Full postconditions
- B.6 Observable invariants

C: Constraints

D: Design

E: Extension

(From lesson 1) Style rules

No routine without header comments

Preconditions always fully expressed

Counter-example!

Postconditions and invariants: the more the better

Redundancy OK in class invariants (axioms *and* theorems)

Standardized layout

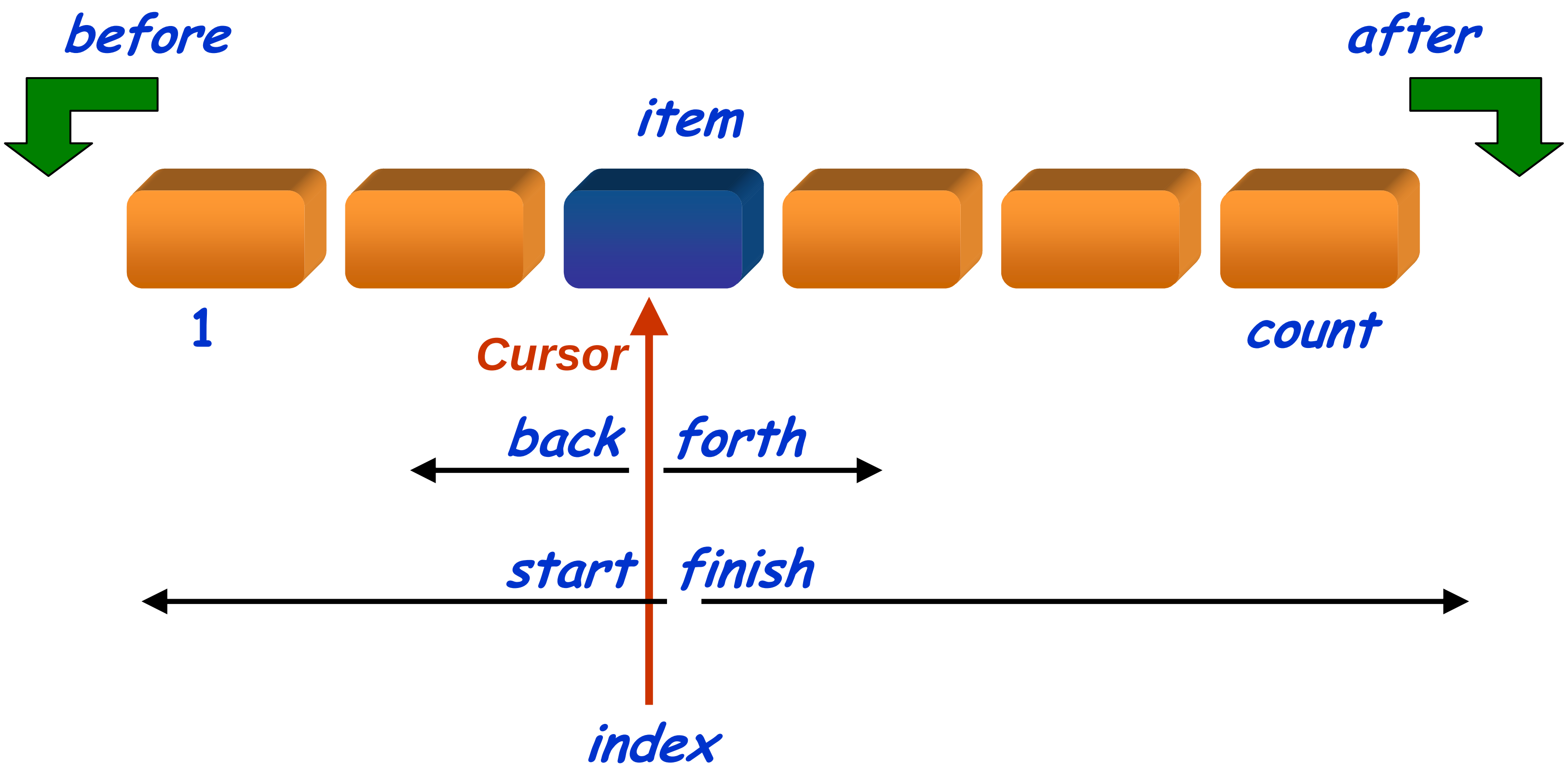
Queries never use verbs!

Class *ACCOUNT*:
balance, not *get_balance*

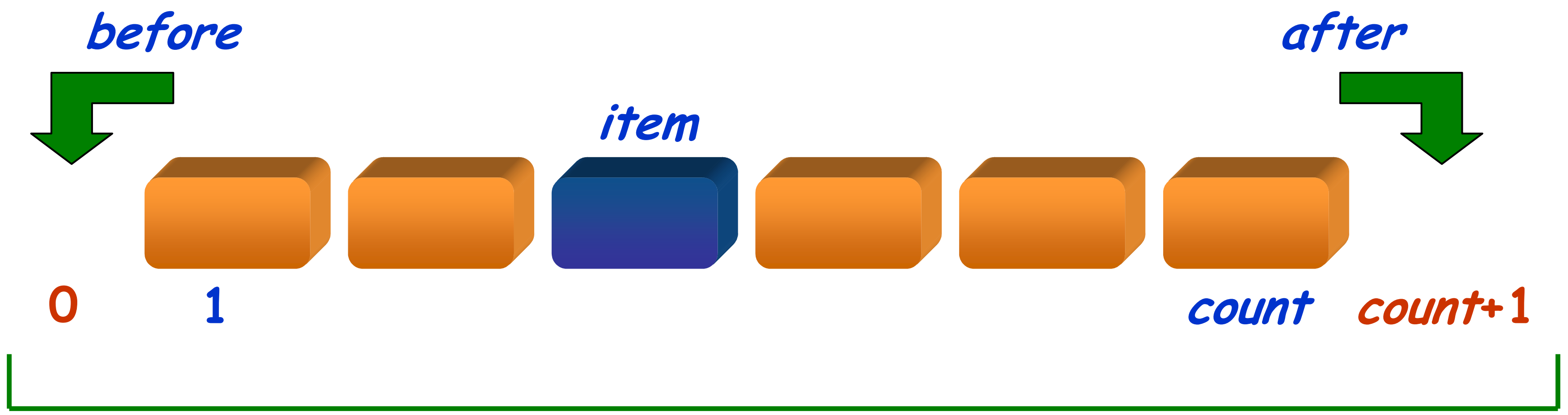
Systematic naming conventions

No exceptions; rules strictly enforced

A list with its cursor



Lists with cursor



Valid cursor positions

From the invariant of class *LIST*

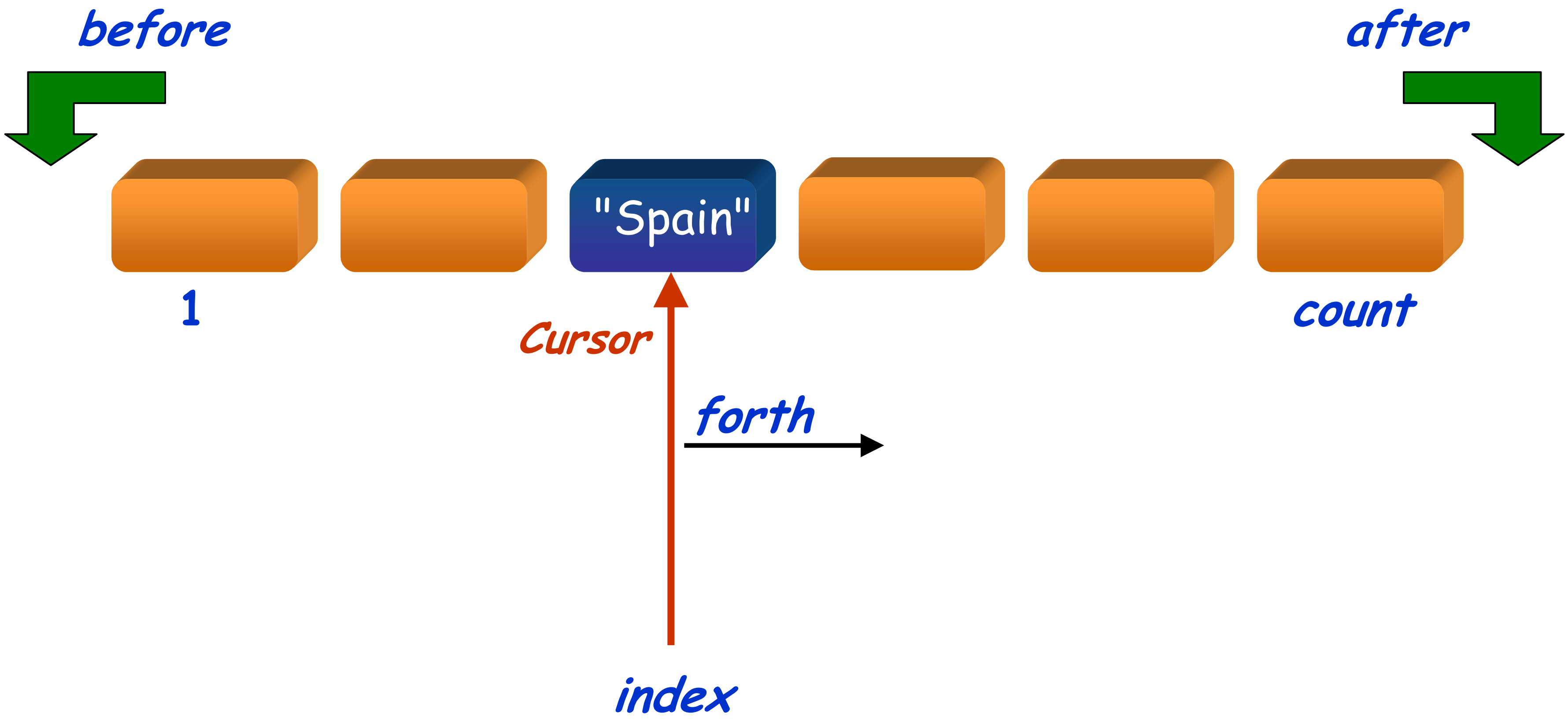
Editor

```
invariant

    prunable: prunable
    before_definition: before = (index = 0)
    after_definition: after = (index = count + 1)
        -- from CHAIN
```

Valid cursor positions

Moving the cursor forward



ETH Software Engineering

Specifying a command: *forth*

```
feature -- Status report

    after: BOOLEAN
            -- Is there no valid cursor position to the right of cursor?

    before: BOOLEAN
            -- Is there no valid cursor position to the left of cursor?

feature -- Cursor movement

    forth
            -- Move to next position; if no next position,
            -- ensure that `exhausted' will be true.
```

What we do with contracts today

- Specify, design, implement

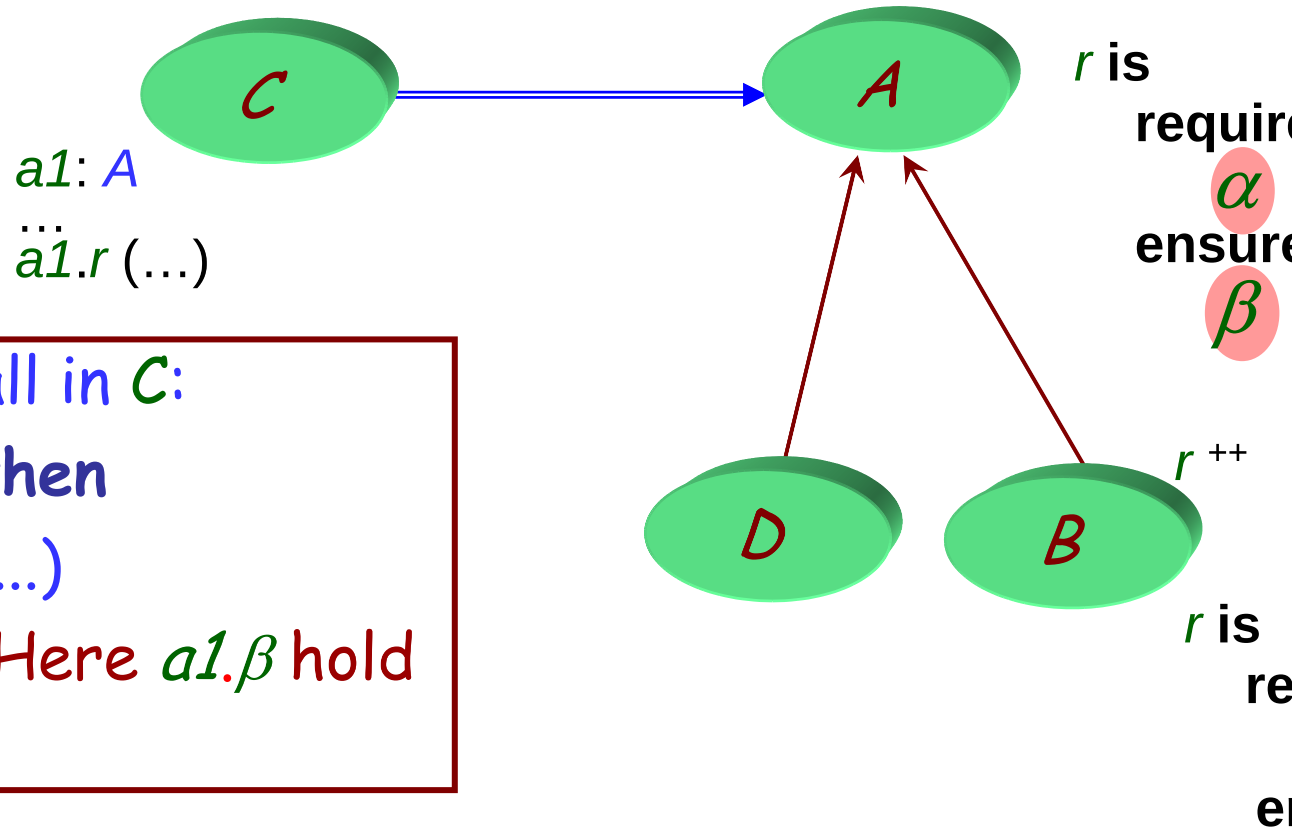
  "Methodology" : the opposite of Defensive Programming

- Document
- Test & debug

  Demo

- Control inheritance, exceptions
- Manage

Contracts and inheritance



Correct call in C:

if *a1*.α **then**

a1.*r* (...)

-- Here *a1*.β hold

end

Client

Inheritance

++ Redefinition

Assertion redeclaration rule

When redeclaring a routine, we may only:

- Keep or weaken the precondition
- Keep or strengthen the postcondition

Assertion redeclaration rule in Eiffel

A simple language rule does the trick!

Redefined version may have nothing (assertions kept by default), or

```
require else new_pre
ensure then new_post
```

Resulting assertions are (approximately):

- *original_precondition* **or** *new_pre*
- *original_postcondition* **and** *new_post*

The next step

Prove that class implementations satisfy the contracts

Principles

Very simple mathematics only

- Logic
- Set theory
- Explainable to a first-year student

Have as few instances of "*Deus ex machina*" (also known as "*pulling a rabbit out of a hat*") as possible

[Physicists: constants
Mathematicians: axioms]

Target and scope

This work applies to Eiffel components

No claim of applicability to any other environment

“Eiffel” may mean either

- Eiffel
- Whatever we need it to be

Computer science is not a natural science

Challenges

- Dealing with a full-fledged, useful, used language
- Loops
- Pointer (reference) structure, dynamic aliasing
- Genericity
- Inheritance, single and multiple
- Polymorphism
- Dynamic binding
- Exception handling
- Agents (routine objects)

Rest of today

Some assets

- Contract mechanism is built-in
- No in-class overloading
- Simple language (e.g. just one form of loops)
- Strict command-query distinction
- Good libraries, extensively reused, contract-rich
- Every loop is characterized by an invariant and a variant (no need for fixpoints etc.)

Some of our friends

Binary relations $A \leftrightarrow B$

Set of pairs $[a, b]$ with $a \in A, b \in B$

A is source set and B is target set

Functions (possibly partial) $A \rightarrow\!\!\!\!\!+ \; B$

(finite) $A \twoheadrightarrow\!\!\!\!\!\!+\!\!+ \; B \subseteq A \rightarrow\!\!\!\!\!+ \; B$

(total) $A \rightarrow B \subseteq A \rightarrow\!\!\!\!\!+ \; B$

For any relation r : **domain** r , **range** r

Function application: $r(a)$, where r is a function and $a \in$ **domain** r

Even if $r : A \leftrightarrow B$ is not a function, we may use image

$r\{X\}$ where $X \subseteq A$ (then $r\{X\} \subseteq B$)

A desirable mode of reasoning

-- SOME_PROPERTY holds of *a*

"Apply SOME_OPERATION to *b*"

-- SOME_PROPERTY still holds of *a*

Applicable to "expanded values", e.g. integers:

-- $P(a)$

$b := b + 1$

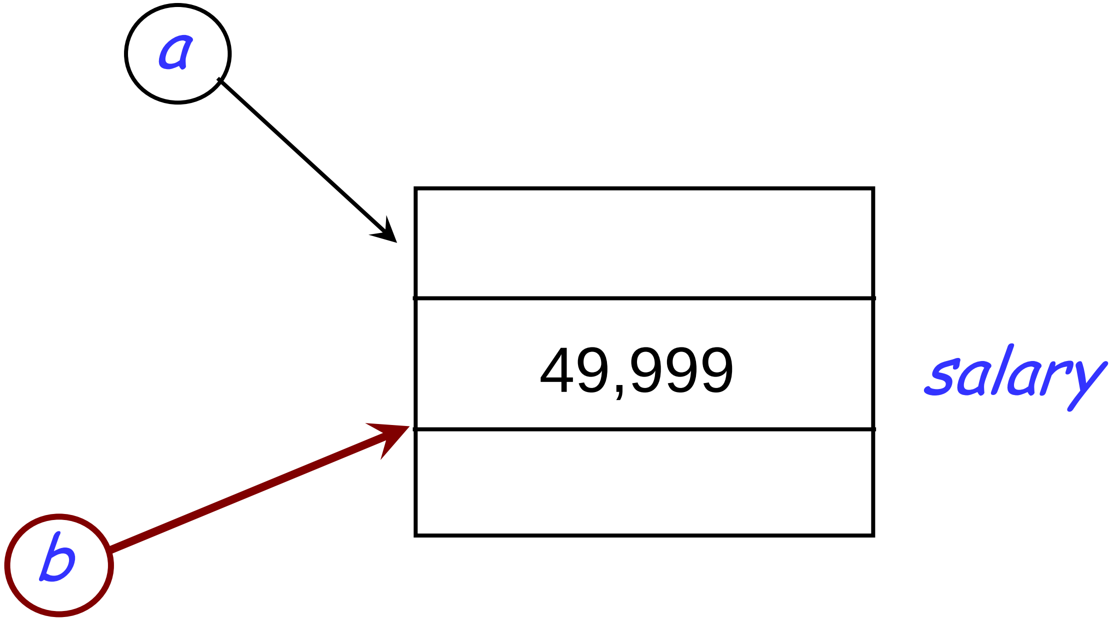
-- $P(a)$

Reasoning in the presence of aliasing

-- *a* makes less than 50 K

b.raise_salary (1)

-- What about *a* ?



This is not just a programming problem

-- I heard that one of the CEO's in-laws makes less than 50K

Memo to personnel: Raise Jill's salary by one euro

-- ?

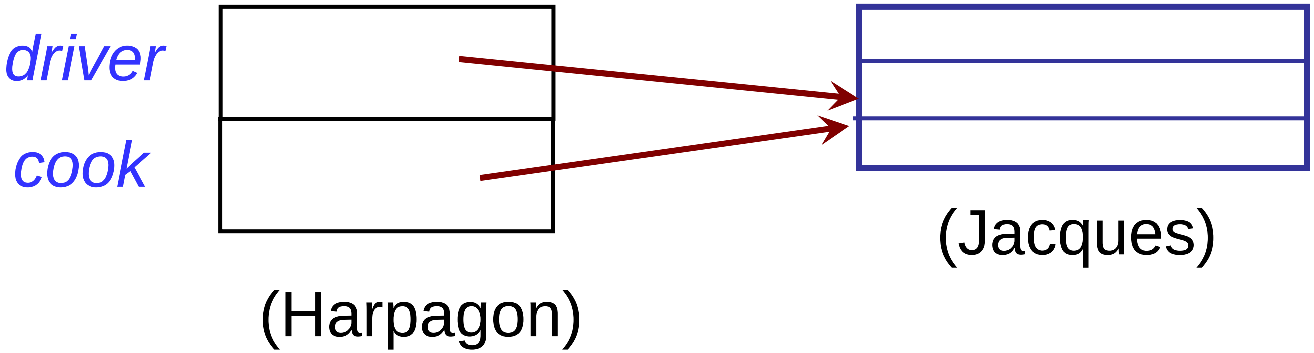
Metaphors cause dynamic aliasing

"The beautiful daughter of Leda"

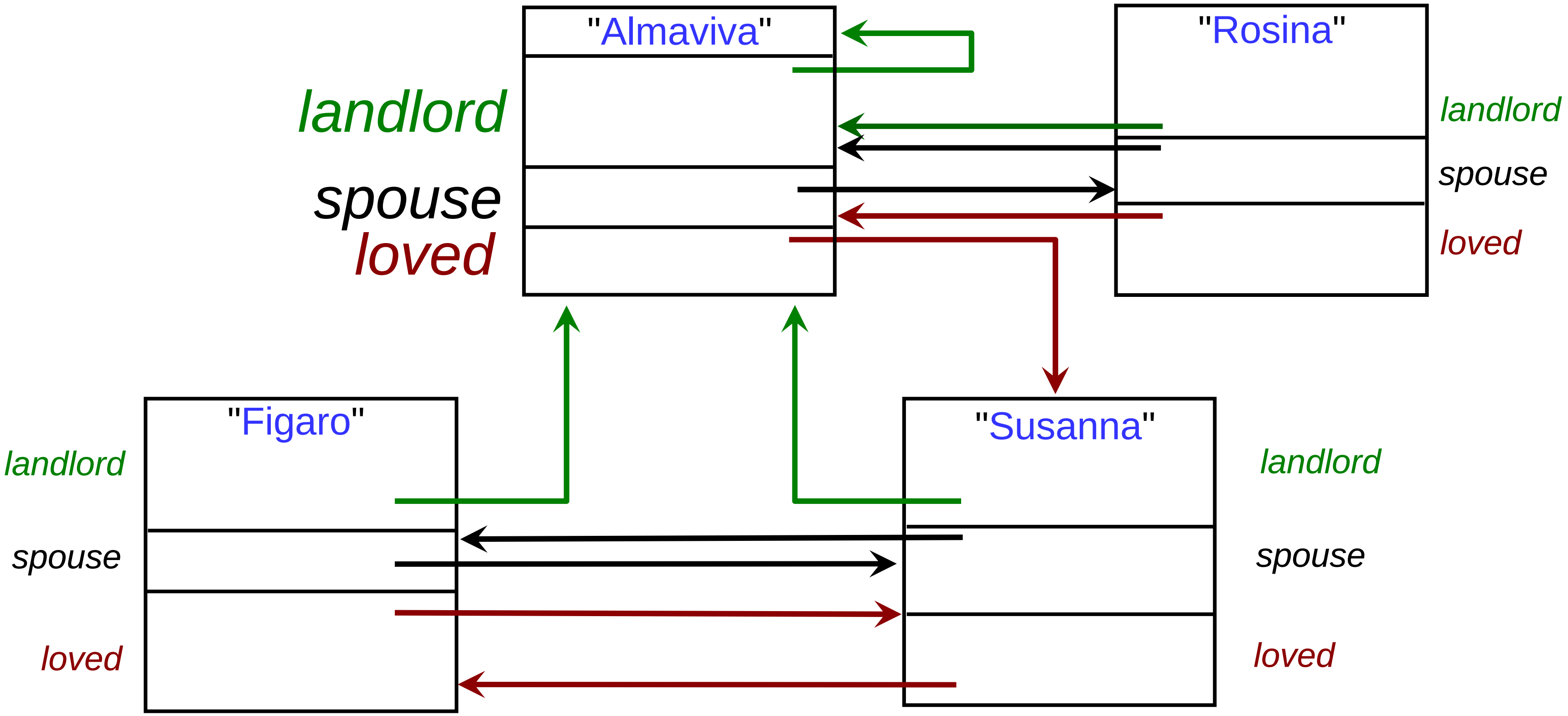
"Menelas's spouse"

"Paris's lover"

"Your driver or your cook?" (Harpagon, in *The Miser*)



Handling pointers



Basic sets

Addresses

-- (Abstract) addresses of potential objects

States

-- Possible computation states

Convention: the name of a set always starts with an upper-case letter. It is either:

- A noun in the plural, suggesting the set's elements
  Example: *States* .
- A noun in the singular, or an adjective, suggesting the set as a whole
  Examples: *Heap, Live*

The objects in a state

$allocated: States \rightarrow \mathscr{P}(Addresses)$

Powerset

-- Set of addresses allocated to objects

The static picture

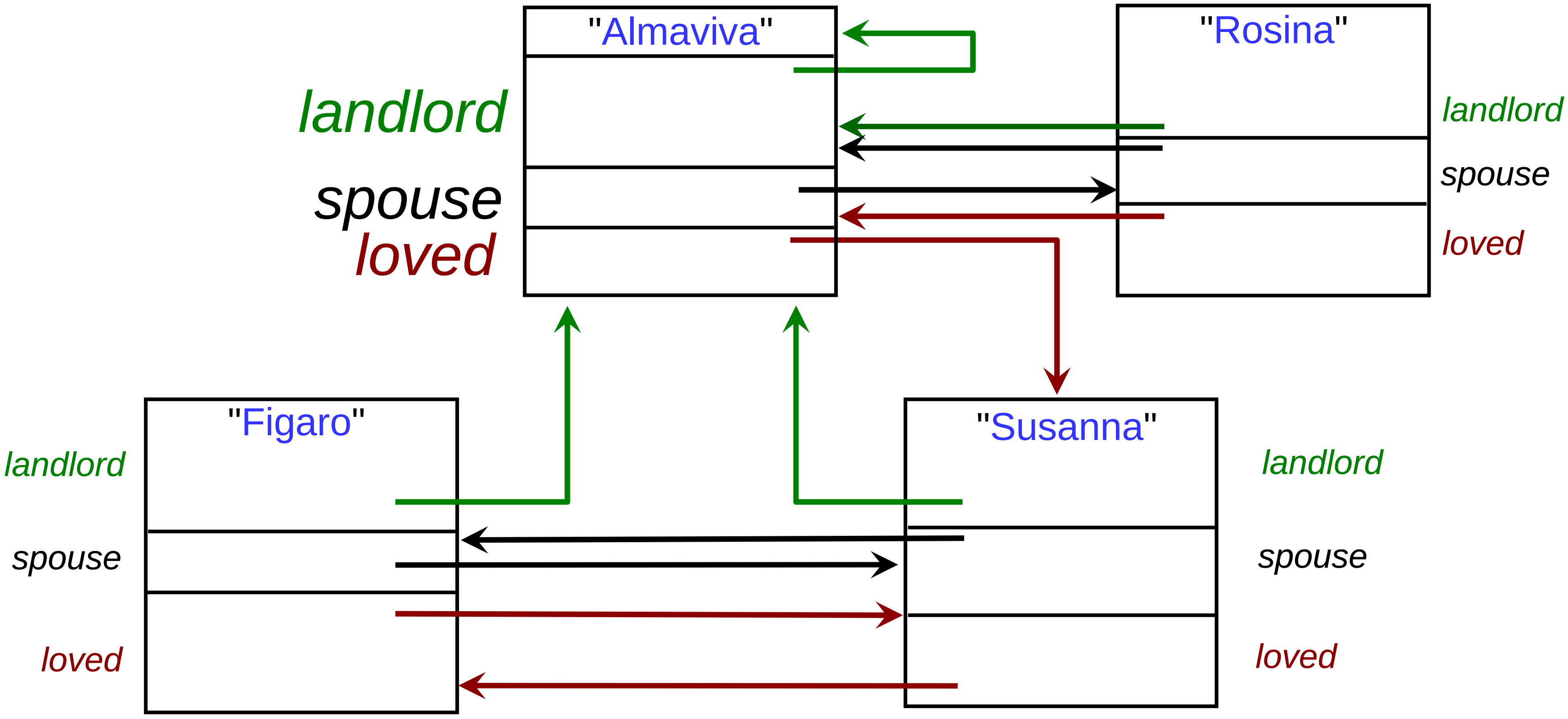
In the next part of the discussion we focus on one specific state s, and define

$$Objects \triangleq allocated(s)$$

"Is defined as"

Modeling attributes

landlord : *States* → *Objects* ⇸ *Objects*

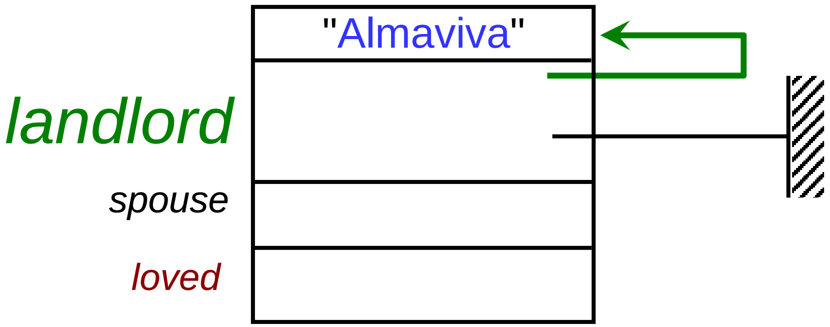


Using partial functions

landlord : *States* → *Objects* ⇸ *Objects*

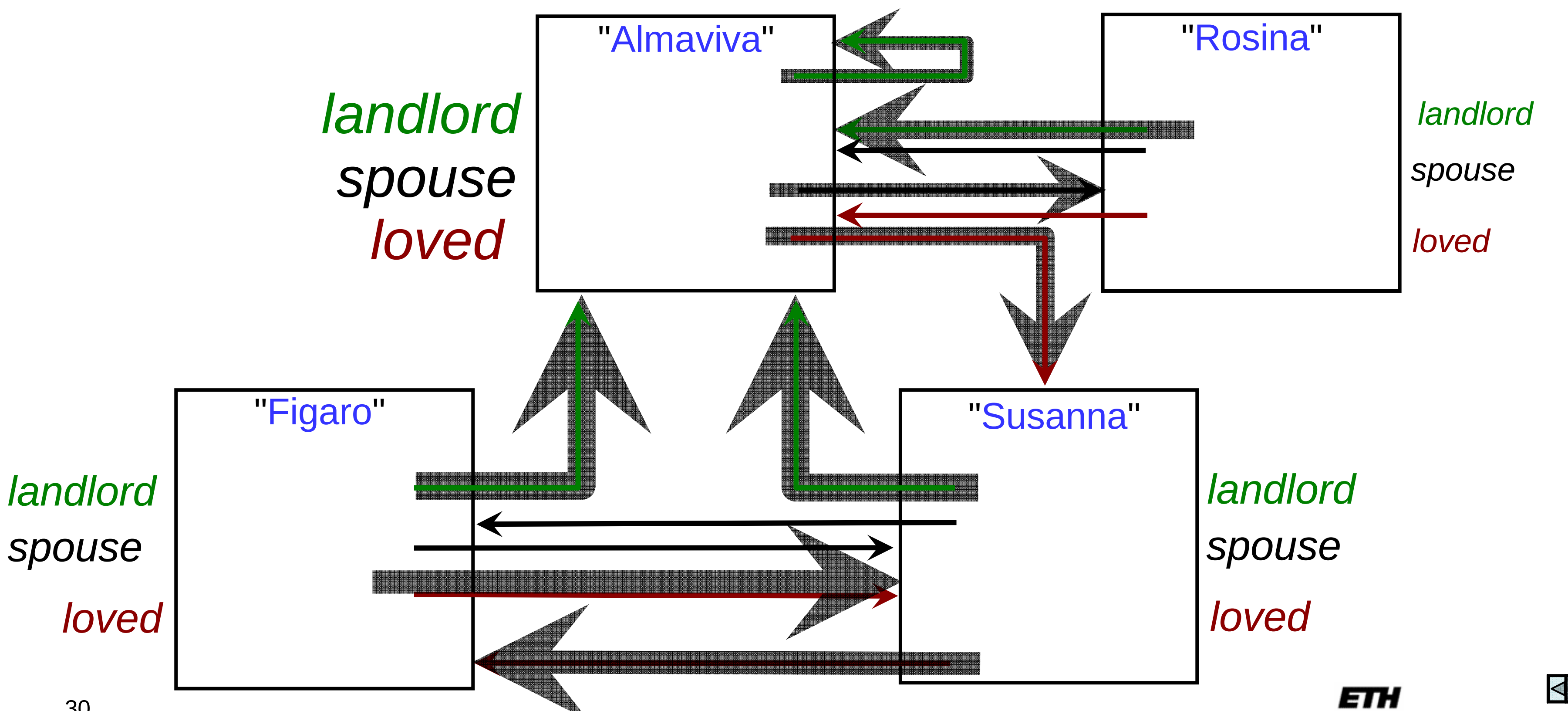
An undefined value for *landlord* (*s*) (*obj*) may signal:

- That the function is not applicable to *obj* (wrong type)
- That the reference exists but is void

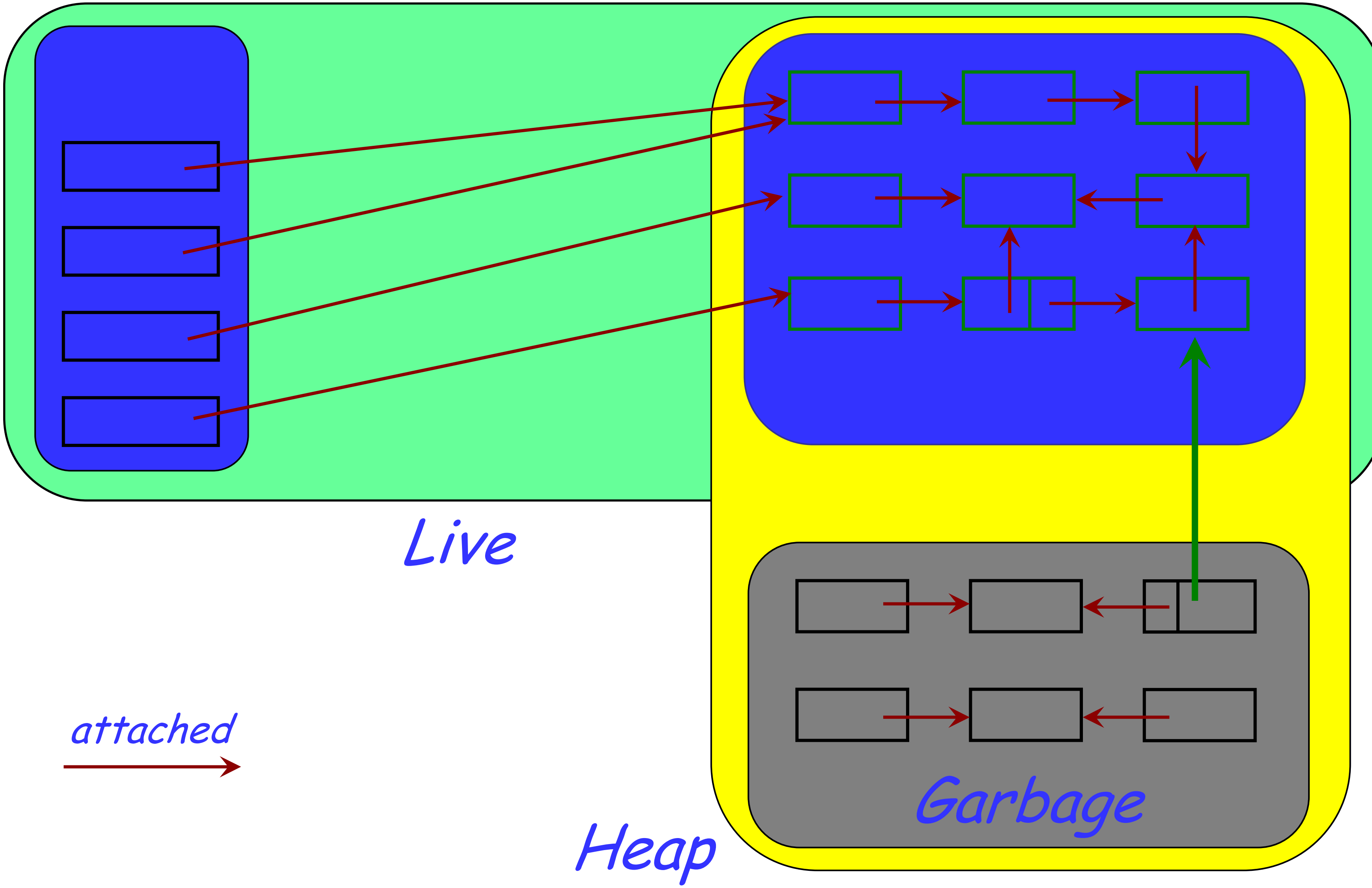


The overall reference relation

$$attached = \bigcup_{a \in attributes} a$$



Parts of the object store



Modeling the pointer relation

$$attached : Addresses \leftrightarrow Addresses$$

Invariant (Basic Object Constraint):

[BOC] $attached \subseteq Objects \leftrightarrow Objects$

Theorems (immediate consequences of [BOC]):

range $attached \subseteq Objects$ -- No zombies

domain $attached \subseteq Objects$ -- No big brother

The stack: source of all references

$Stack : \mathcal{P}\ (Addresses)$

Invariant:

[IS] $Stack \subseteq Objects$ − **range** *attached*

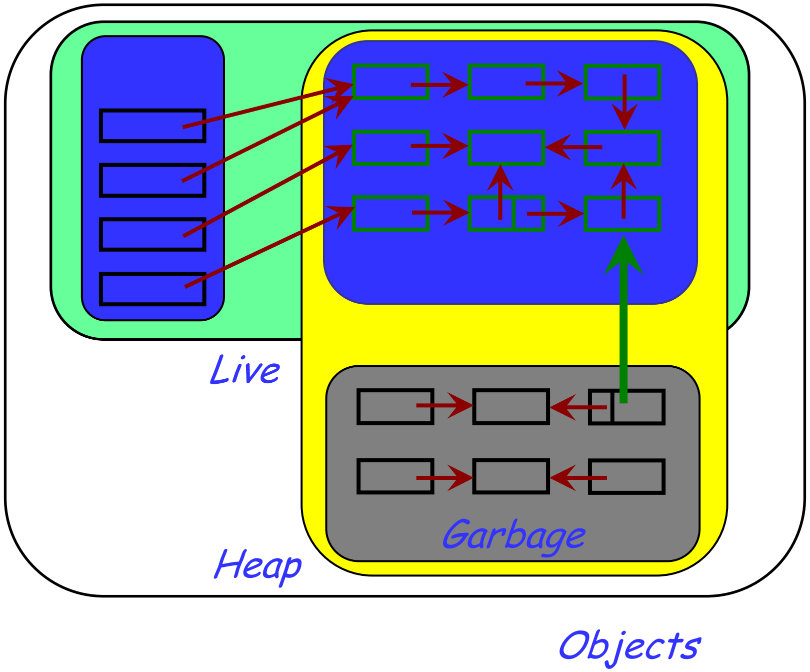
Would not hold in e.g. C++

Live objects, garbage

$Heap \triangleq Objects - Stack$

$Live \triangleq attached^* \{Stack\}$

$Garbage \triangleq Objects - Live$



Some theorems

$Stack \subseteq Objects - \mathbf{range}\ attached$

$Stack \subseteq Live$

$Heap \cap Stack = \varnothing$

$Objects = Stack \oplus Heap$

$Attached\{Objects\} \subseteq Heap$

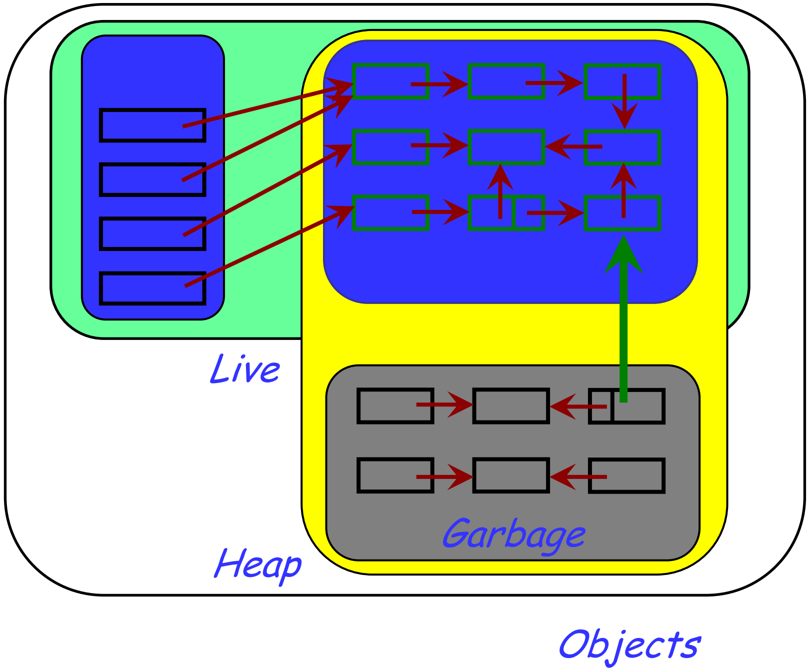
$Attached^{+}\{Objects\} \subseteq Heap$

$Attached\{Live\} \subseteq Live$

$Attached^{*}\{Live\} \subseteq Live$

$\mathbf{range}\ attached \subseteq Heap$

$Objects = Stack \oplus (Live - Stack) \oplus Garbage$



Example events

- Object creation
- Incremental garbage collection
- Full garbage collection

All must preserve invariants!

Convention for state transformers ("events")

$t : State \times V \nrightarrow State$

Queries such as $q, r : State \rightarrow Some\_type$

Notation:

$t\,(v : V)$ **is**

require

$some\_property\,(s, v)$

do

$q :=$ ...

$r :=$ ...

ensure

$rel\,(q,$ **old** $q)$

end

Specifies domain

Effect (in parallel) on specific queries; others unchanged!

$\forall\ s, v :$
$rel\,(q\,(s), q\,(t\,(s, v)))$

Equality

- We use = throughout, but mean object equality: ~

- **same** x, in a postcondition, means

  x ~ **old** *x*

Example event: object creation (1, spot mistake!)

allocate (*existing*, *new* : *Addresses*) **is**

    -- Allocate new object at *new*, linked from *existing*.

require

    old_exists: $existing \notin Live$

    new_unused: $new \in Live$

do

    $Objects := Objects \cup \{new\}$

    $attached := attached \cup \{[existing, new]\}$

ensure

    possibly_one_more: $Objects =$ **old** $Objects \oplus \{new\}$

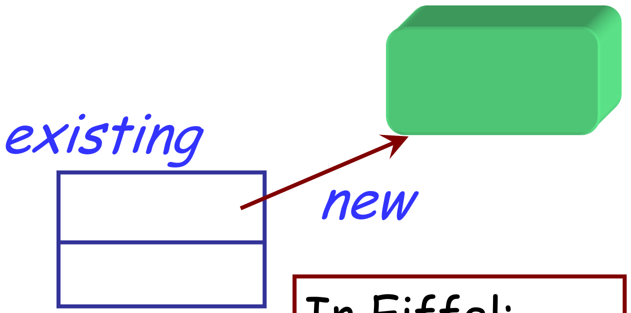
    new_reachable: $Live =$ **old** $Live \{new\}$

    other_garbage_remains: **same** $(Garbage - \{new\})$

    no_change_to_stack: **same** $Stack$

    rest_unchanged: **same** $(attached \{Objects - \{new\}\})$

end

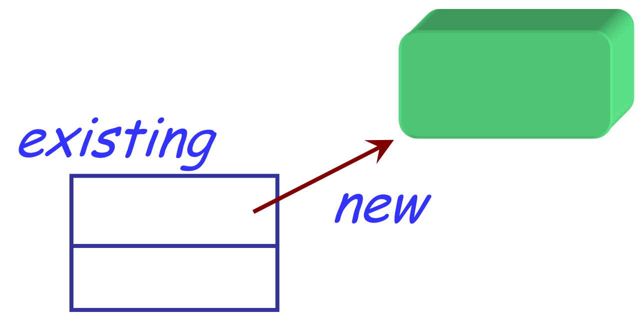


In Eiffel:
    create *new*

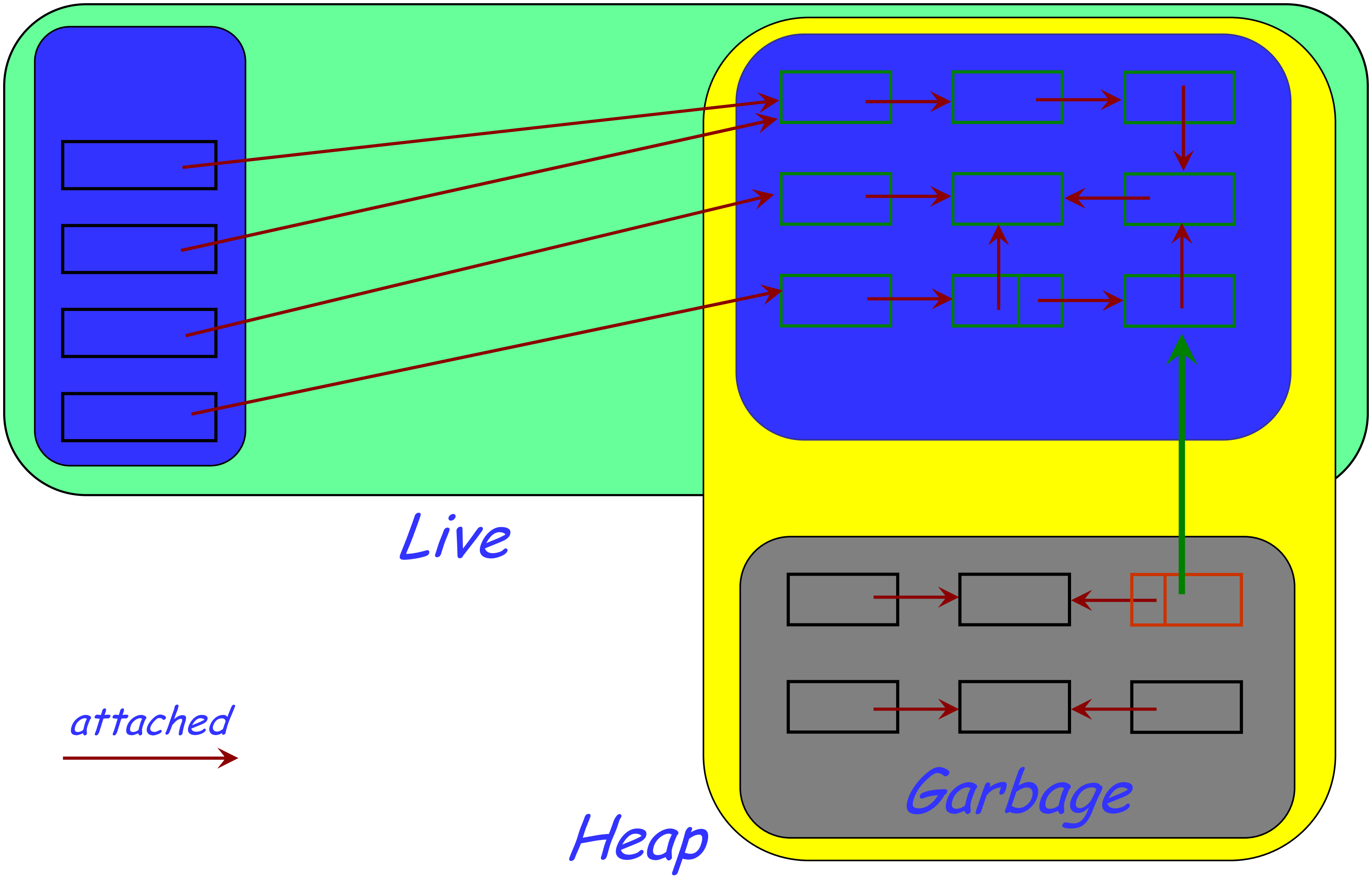
Example event: object creation (1, spot mistake!)

```
allocate (existing, new : Addresses) is
        -- Allocate new object at new, linked from existing.
    require
        old_exists:    existing ∉ Live
        new_unused:  new         ∈ Live

    do
        Objects   := Objects  ∪ {new}
        attached  := attached ∪  {[existing, new]}
    ensure
        possibly_one_more:    Objects = old Objects ⊕ {new}
        new_reachable:          Live = old Live ∪ {new }
        garbage_remains:       same (Garbage – {new})
        no_change_to_stack:  same Stack
        rest_unchanged:         same (attached {Objects – {new}})
    end
```



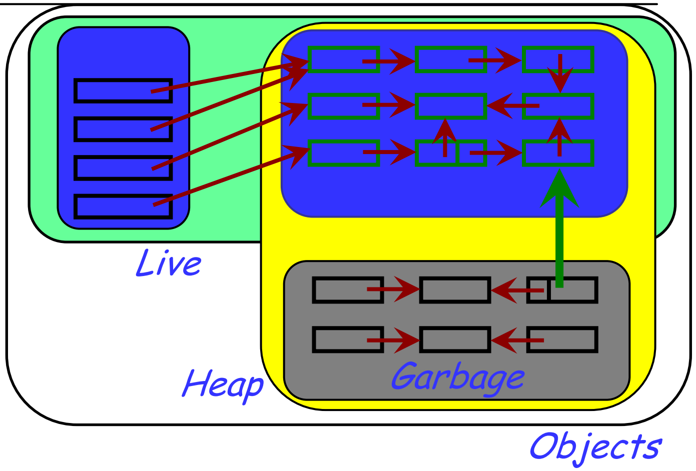
Object structure



Example event: object creation (2, correct!)

```
allocate (existing, new : Addresses) is
        -- Allocate new object at new, linked from existing.
    require
        old_exists:     existing ∉ Live
        new_unused:     new      ∈ Live
        new_virginal:   new      ∉ domain attached
    do
        Objects  := Objects  ∪ {new}
        attached := attached ∪ {[existing, new]}
    ensure
        possibly_one_more:    Objects = old Objects ⊕ {new}
        new_reachable:        Live = old Live {new }
        garbage_remains:      same (Garbage – {new})
        no_change_to_stack:   same Stack
        rest_unchanged:       same (attached {Objects – {new}})
    end
```

Garbage collection, full




```
collect_all is
        -- Get rid of all garbage objects.
    do
        Objects := Live
        attached := attached \ Live
    ensure
        live_only:            Objects = old Live
        restricted_to_live:   attached = old (attached \ Live)
        restricted_to_kept:   attached = (old attached) \ Objects
        no_change_to_stack:   same Stack
        no_loss_of_life:      same Live
        all_from_live:        domain attached ⊆ Live
        all_to_live:          range attached ⊆ Live
        all_live:             Objects = Live
        garbage_removed:      Garbage = ∅
    end
```

Restriction

Garbage collection, incremental

```
collect_some (Rejects: (Objects)) is
        -- Get rid of all the objects in Rejects.
  require
        recyclable: Rejects ⊆ Garbage
  do
        Free := Free ∪ Rejects
        attached := attached \ Rejects
  ensure
        restricted_to_kept:               attached = (old attached) \ (Objects – Free)
        no_change_to_stack:               same Stack
        no_loss_of_life:                  same Live
        from_live_or_garbage: domain attached ⊆ Live ∪ (Garbage – Rejects)
        all_live_or_free_or_garbage: Objects = Live ∪ Free ∪ (Garbage – Rejects)
        no_change_to_garbage:             Garbage = old Garbage
        no_change_to_objects:             Objects = old Objects
        possibly_more_free:               old Free ⊆ Free
  end
```

Garbage collection, incremental

```
collect_some (Rejects: (Objects)) is
        -- Get rid of all the objects in Rejects.
  require
        recyclable: Rejects ⊆ Garbage
  do
        Free := Free ∪ Rejects
        attached := attached \ Rejects
  ensure
        restricted_to_kept:            attached = (old attached) \ (Objects – Free)
        no_change_to_stack:            same Stack
        no_loss_of_life:               same Live
        from_live_or_garbage: domain attached ⊆ Live ∪ (Garbage – Rejects)
        all_live_or_free_or_garbage: Objects = Live ∪ Free ∪ (Garbage – Rejects)
        no_change_to_garbage:          Garbage = old Garbage
        no_change_to_objects:          Objects = old Objects
        possibly_more_free:            old Free ⊆ Free
  end
```

Summary of lesson 2

"Proving a class" means proving that it satisfies its contracts

A simple theoretical framework seems sufficient: sets, relations, total and possibly partial functions.

To make proofs convincing we should avoid special notations

We can express complete specifications through models

Reference attributes can be modeled through functions

The overall pointer structure can be modeled through a relation, the union of these functions

We can effectively model the object store and events such as garbage collection