Proving Pointer Program Properties

Part 2: *The overall object structure*

Bertrand Meyer

ABSTRACT

The run-time object structure of object-oriented programs typically relies on extensive use of references (or *pointers*). This second part of a general mathematical framework for reasoning about references handles the overall properties of the structure, not distinguishing between individual links but only considering whether *any* reference exists between two objects. It provides a basis for dealing with memory management and especially garbage collection.

This is part of a series of articles. See here for links to the others.

2.1 BASICS OF THE RELATION MODEL

The coarse-grained model of object structures developed here will rely on a *relation* between objects. For this reason we call it the relation model; the finer-grained models of the subsequent articles, which take into account individual attributes of classes, and hence individual fields of objects, will expand this relation into a set of *functions*.

Addresses and objects

The execution of an object-oriented system creates and manipulates objects. Each object is stored at a certain address, and we can only use a finite set of addresses. We express this by introducing a constant, the set of addresses:

> [A1]   *Addresses***:** $\mathbb{F}$ ($\mathbb{N}$)   ●

where $\mathbb{N}$ is the set of natural integers and $\mathbb{F}$ *(X)*, for any set *X*, is the set of all finite subsets of *X* (a subset of $\mathbb{P}$ *(X)*, the powerset of *X*, which contains all its subsets). Property [A1] specifies *Addresses* as a finite set of integers.

   This property is an *axiom* asserting the existence of a distinguished member, *Addresses*, of a known set $\mathbb{F}$ ($\mathbb{N}$); the colon "**:**" recalls the type declarations of some programming languages.

   An element of *Addresses* represents a potential object; only at certain addresses will we find actual objects. We represent this observation by introducing an explicit variable subset of *Addresses*:

> [A2]   *Objects***:** $\mathbb{F}$ ($\mathbb{N}$)   ●

with the property that

> [I3]   *Objects* $\subseteq$ *Addresses*   ●

[I3] is an *invariant*; we will have to prove that every event preserves it.

> A point of modeling style, also applicable to later axioms and invariants: it would be possible to avoid the invariant [I3] altogether by defining the set *Objects* directly, in the axiom [A2], as a member of $\mathbb{F}$ (*Addresses*) rather than $\mathbb{F}$ ($\mathbb{N}$). The practical effect is the same: we would still have to prove that any element added to *Objects* by any event is in *Addresses*; this is a type check rather than an invariant preservation proof. Making the invariant explicit is clearer.

The choice of natural integers — $\mathbb{F}$ ($\mathbb{N}$) — for *Addresses*, and as a consequence for *Objects*, deserves a justification. It is legitimate to restrict ourselves to a finite set of addresses since this is what we will have on any actual computer or bank of computers; representing them by integers does not imply any implementation commitment but simply reflects the concept known as **object identity**: each object created during execution has a separate identity, even if its content happens to be identical to that of another object. Each element of *Addresses* represents such a unique identity for a possible object; it does not have to represent a physical address in the memory of a computer. We will explore <u>later</u> the more precise properties of *Addresses* and its possible relation to actual memory addresses.

It will be useful to give a name to addresses not occupied by objects:

> [D4] *Unused* ≜ *Addresses– Objects*

where – is set difference. (≜ means "is defined as.")

Linking objects

The topic of our study is the set of reference links that may exist between objects. At the highest level of abstraction, we represent it by a relation

> [A5] *attached*: $\mathbb{N} \leftrightarrow \mathbb{N}$

where $A \leftrightarrow B$ is the set of relations between any two sets A and B. (More precisely we are interested in $\mathbb{N} \leftrightarrow\!\!\!\!\!\rightarrow \mathbb{N}$, the set of finite relations, but this makes no difference since *Addresses* itself is finite, so all relations on it are finite.)

> Names of **sets** of addresses and objects, as *Addresses* and *Objects*, start with an upper-case letter; names of **functions**, **relations** and **predicates**, as *attached*, start with a lower-case letter.

The informal meaning of *attached* is that it contains a pair [*o1, o2*] to reflect that there is a reference from the object at *o1* to the object at *o2*. The name *attached* reflects the Eiffel terminology, which says that at run time a reference may be "attached to" a certain object.

This model — called from now on the **Relation Model** — provides a good basis for our study. At run time your program has created a set of objects. Each object, stored at a certain address, is made of a number of fields; a field may be of an "expanded" type, meaning that it is a directly usable value — an integer, a character … — or it may be a reference. A reference either takes us to another object or is "Void":
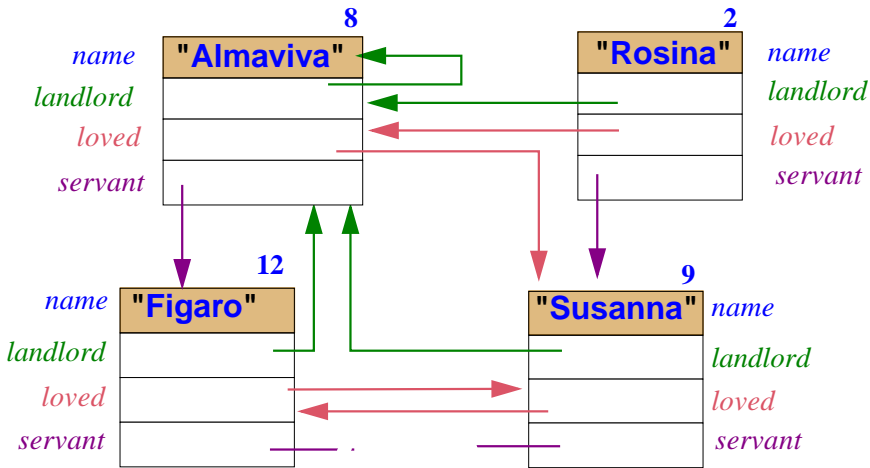
From an object store

Since this discussion focuses on the references, we ignore the actual object contents (the expanded fields). The relation *attached* defines a graph of references between objects. Ignoring individual fields, it only indicates whether at least one reference exists between any two objects; it is represented, for the object graph of the preceding figure, by the bold lines on the next figure. If two fields of an object are references to the same object (as *landlord* and *loved* from the object labeled **2** to the object labeled **8**), they yield only one link in the relation.
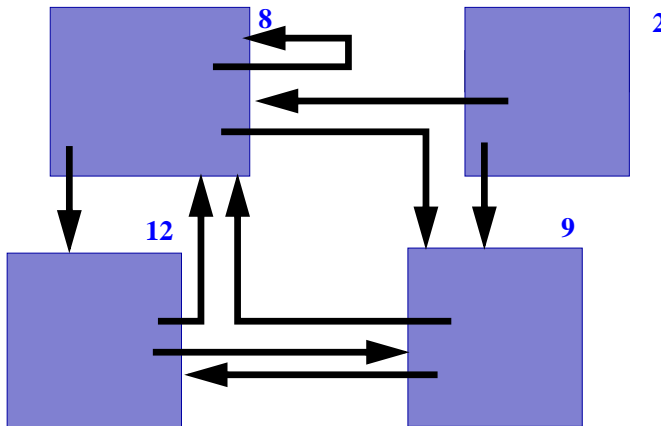
References collapsed into a relation: *attached*

By considering only the relation *attached*, we disregard individual attributes such as *landlord* and *loved*; to get a realistic model <u>we will have</u> to reintroduce them. Until then we can use *attached* as a coarse-grain view of the object structure, already sufficient to obtain a remarkable set of properties.

The model defines *attached as* a relation, meaning a set of pairs; that a particular pair of addresses [*i*, *j*] belongs to the relation (in symbols, [*i*, *j*] ∈ *attached*) means that there is a reference from the object at *i* to the object at *j*. With the addresses shown, the relation illustrated is the set of pairs {[8, 8], [8, 9], [8, 12], [2, 8], [2, 9], [12, 8], [12, 9], [9, 8], [9, 12]}.

Void links

Every practical programming language that offers references has a notion of "void" or "null" reference, used in particular to terminate chains of references in linked structures. In Eiffel, *Void* also serves as default initialization value for reference types (like false for *BOOLEAN* and zero for *INTEGER*).

> The Self language tried to do away with *Void*, but the result seems to confirm the need for this concept.

How do we model *Void*? We don't. One of the benefits of using relations and partial functions is to spare us the need for any special element to represent *Void*. If a reference from *obj* is void, the corresponding function, a subset of *attached*, will simply not be defined for *obj* — will not contain any pair of the form [*obj*, *x*] for any *x*. The preceding figure illustrates this for objects **9** and **12** ("**Figaro**" and "**Susanna**") and function *servant*. This convention is all we need; it will be invaluable when we prove properties of data structures in [12].

The Basic Object Constraint

We require the relation *attached* to satisfy a fundamental invariant, the Basic Object Constraint:

> [I6]   *attached* ⊆ *Objects* ↔ *Objects*

●

which expresses that links only exist between objects, not arbitrary addresses. The Basic Object Constraint is the combination of two separate properties:

> [T7]   **domain** (*attached*) ⊆ *Objects*
> [T8]   **range** (*attached*) ⊆ *Objects*

where, if *r* is a relation, **domain** (*r*) is its domain, the set of elements *x* such that *r* contains a pair of the form [*x*, *y*] for some *y*; and **range** (*r*) is its range, the set of *y* such that *r* contains a pair of the form [*x*, *y*] for some *x*.

> The model could define [T7] and [T8] as two independent invariants and deduce the Basic Object Constraint [I6] as a theorem; we choose to do the reverse, taking the more complete property as the invariant.

The first part [T7] of the invariant, **domain** (*attached*) $\subseteq$ *Objects*, states that all references to objects come from other objects: there's no one out there keeping references to our objects. We accordingly call it the **No Big Brother property**. It <u>will be</u> essential for modeling dynamic object *allocation*, a cornerstone of programming with dynamic data structures. Any event that creates an object must be able to use any unallocated memory address, meaning any element of the set *Unused* (defined above as *Addresses – Objects*). Without the No Big Brother property such an element could contain links to objects; making it part of *Objects* might then add spurious reference links to the object structure, destroying its consistency and causing trouble for memory management, especially garbage collection.

Maintaining this property invariant will impose a constraint on memory *deallocation* as performed by a garbage collector (GC): when reclaiming a member of *Objects* to return it to *Unused*, the GC will need to erase (or "zero out") all its outgoing links, to satisfy this clause and enable a later memory allocation event to reuse it without risk.

The second part [T8] of the Basic Object Constraint, **range** (*attached*) $\subseteq$ *Objects*, states that if, from an object, we follow a reference, we get an object. We'll call it the **No Zombie property**, using the definition

> [D9]   *Zombies* $\triangleq$ **range** (*attached*) *– Objects*

The Basic Object Constraint prohibits zombies through the following restatement of [T8]:

> [T10] *Zombies* $= \varnothing$

The Basic Object Constraint must be our obsession when we write software in languages meant for manual memory management such as C, Pascal, Ada and C++, where the definition of disaster is to end up with a Zombie object that is referenced by other objects but not known any more as a member of the community of objects. This happens because at some point our program has "freed" the memory allocated to the object (or "disposed" of it in Pascal terminology) even though some non-zombie, somewhere, somehow, still keeps a reference to it.

> A number of companies exist primarily to provide tools that help developers debug programs (often in C and C++) that do not satisfy the Basic Object Constraint, so that promoting that constraint might in the current market conditions appear anti-business. Any advances reported here are, fortunately, of a preliminary nature only.

In a language supported by automatic memory management, such as Eiffel, the Basic Object Constraint is also an obsession, but sensibly transferred from the application programmers to the authors of the memory management system, especially the *garbage collector* (GC).

> The No Zombies property has two other consequences presented later: a particular obligation on <u>*incremental*</u> garbage collection; and, if we understand *Addresses* to represent actual memory addresses, a rejection of references to subobjects, <u>discussed</u> when we look at how to provide a concrete interpretation for the set *Addresses*.

The Basic Object Constraint defines the fundamental invariant under which the GC will pursue its goal of returning to *Useless* any obsolete elements of *Objects*: in this process, it must create neither Big Brothers nor Zombies.

Rather than [T8] or [T10], we will mostly use the No Zombie property under yet another form

-- Basic Object Constraint:
[T11] *attached* (• *Objects* •) $\subseteq$ *Objects*

where, for any relation *r* and a subset *X* of its source set, r (• *X* •) is the image of *X* under *r*: the set of elements *y* such that *r* contains a pair of the form [*x*, *y*] for some member *x* of *X*. Although equivalent to [T8], this form takes advantage of the image operator, which enjoys such pleasant properties as

-- For any subsets *X* and *Y* of the source set of *r* and *s*:
[T12] $X \subseteq Y \Rightarrow r$ (• *X* •) $\subseteq$ r (• *Y* •))
[T13] $r \subseteq s \Rightarrow r$ (• *X* •) $\subseteq$ s (• *X* •))
[T14] r^{-1} (• $\varnothing$ •) $=$ $\overline{\textbf{domain} (r)}$
[T15] $(r \cup s)$ (• *X* •) $=$ r (• *X* •) $\cup$ s (• *X* •)
[T16] r (• *X* •) $\subseteq$ **range** (r)
[T17] r^{+} (• *X* •) $\subseteq$ **range** (r)
[T18] id [*X*] $\subseteq r \Rightarrow r^{*}$ (• *X* •) $\subseteq$ **range** (r)

and <u>more to come</u>. These use the following notations: $\varnothing$ is the empty set; $\overline{X}$ is the complement of a set *X*; *id* [*X*] is the identity relation on *X*; if *r* is a relation, r^{-1} is the inverse relation of *r*, r^{+} its transitive closure, and r^{*} its reflexive transitive closure.

The discussion will rely extensively on the image operator, which lets us treat a relation as a function from subsets to subsets, and hence take advantage of all the notations and properties of functions, more convenient than those of general relations.

2.2 STACK, HEAP, GARBAGE AND LIVE OBJECTS

In an object store, some objects are "live" and other are "garbage". The live objects are those reachable, directly or indirectly, from "root" objects.

Stack and heap

Following the common structure of O-O programming language implementation, we refer to the set of root objects as "The Stack" and introduce it explicitly as a variable:

[A19] *Stack*: $\mathbb{F}$ ($\mathbb{N}$)   ●

We require the stack to satisfy two properties:

•   Since the model treats all data as objects, the stack may only contain objects. Some of the values on the stack may be very simple, for example a single integer (of no interest at this stage since we ignore non-reference data) or a single reference, but we model them as objects all the same.

•   We exclude any references leading *to* stack objects, although references may exist *from* stack objects to others, called "heap" objects:
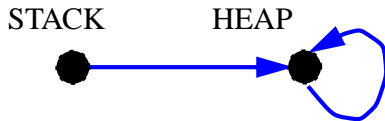
What pointers may point to

(*From* [7]).

The second property does not hold in C++, with its arbitrary C-style pointers, but Eiffel and some other object models observe it because it makes programming simpler and safer.

The combination of these properties is our third invariant:

[I20]   *Stack* $\subseteq$ *Objects* – **range** (*attached*)   ●

Together with [I3], this invariant implies:

[T21] **range** (*attached*) $\cap$ *Stack* $\subseteq$ $\varnothing$
[T22] *Stack* $\subseteq$ *Objects*
[T23] *Stack* $\subseteq$ *Addresses*

We may now define the **heap**:

[D24] *Heap ≜ Objects – Stack*

implying that

[T25] *Heap ⊆ Objects*
[T26] *Heap ∩ Stack = ∅*
[T27] *Objects = Stack ⊕ Heap*
[T28] **range** (*attached*) ⊆ *Heap*

Here ⊕ denotes disjoint union of subsets; a property of the form $A \oplus B = C$, such as [T27], stands for two separate properties: $A \cap B = \varnothing$ and $A \cup B = C$. [T27] follows from [D24] and [T22]. Together with the No Zombies theorem [T8], [T27] implies **range** (*attached*) ⊆ *Stack ⊕ Heap*, from which [T21] yields the theorem [T28]. This theorem explains the importance of the heap.

Using the image operator we may draw a set of consequences from the preceding properties:

[T29] *attached* (• *Objects* •) ⊆ *Heap*
[T30] *attached* (• *Heap* •) ⊆ *Heap*
[T31] *attached*$^{+}$ (• *Objects* •) ⊆ *Heap*
[T32] *attached** (• *Heap* •) ⊆ *Heap*

[T29] uses [T16] to restate [T28]. [T30] follows from [T29] through the subsetting property [T12] of images. The last two theorems follow from the previous two by iterating the same reasoning.

This proof of [T31] and [T32] is a fixpoint proof (generalized proof by induction) in the following sense: if for a certain predicate *p*, a certain relation *r* and a certain set *A* we can show that *p* (*A*) holds, and moreover that whenever *p* (*X*) holds *p* (*r* (• *X* •)) also holds, then we may deduce that *p* (*r** (• *A* •)) holds. This is also how [T17] and [T18] follows from [T16].

Live and garbage objects

From roots (the stack) we define **live objects**, those reachable from the stack directly or indirectly:

> [D33] *Live* ≜ *attached** (· *Stack* ·)

as illustrated by the following informal figure of the object store, whose details will soon be completely clear.

The overall structure
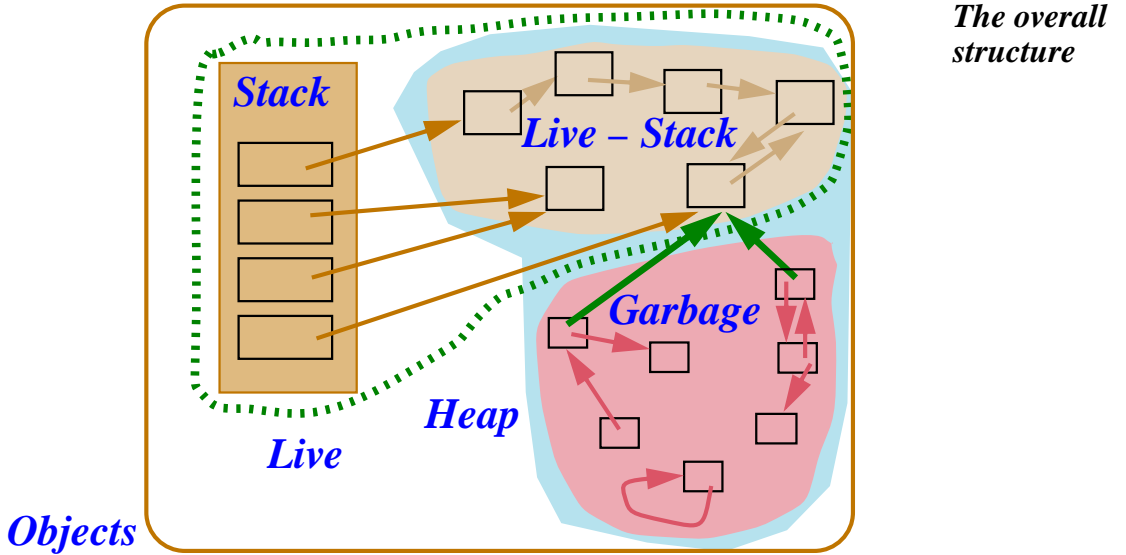
Some properties follow immediately from the definition of *Live* [D33]:

> [T34] *Stack* ⊆ *Live*
> [T35] *Live* ⊆ *Objects*
> [T36] *attached* (· *Live* ·) ⊆ *Live*
> [T37] *attached** (· *Live* ·) ⊆ *Live*

[T34] comes from the general property of closures that *id* [X] ⊆ r* for any relation *r* of source set *X*. [T35] results from the combination of [T22], the closure property of images [T18], and the No Zombie property [T8]. [T36] follows from taking the image by *attached* of both sides of [D33] and using the general property that *r* composed with *r** is a subset of *r**. [T37] takes [T36] to its fixpoint using [T18].

Outside of *Live* we find the non-reachable objects, known more prosaically as **garbage** (hence "Garbage Collector"):

[D38] *Garbage* ≜ *Objects – Live*

sometimes more convenient to use under the form

[T39] *Objects = Live ⊕ Garbage*

which follows from [T35], and itself implies

[T40] *Garbage ∩ Stack = ∅*
[T41] *Garbage ⊆ Heap*
[T42] *Heap = (Live – Stack) ⊕ Garbage*
[T43] *Objects = Stack ⊕ (Live – Stack) ⊕ Garbage*
[T44] *attached*$^{-1}$ (• *Garbage* •) ⊆ *Garbage*

Again you may follow all these properties on the preceding figure. [T40] is a consequence of [T34] and [D38]. [T41] restates [T40] based on [D24]. [T42] restates [D24] using [T39] and [T40]. Then [T43] combines [T27] and [T42]. [T44] states that links into *Garbage* may only come from *Garbage*; this is a consequence of [T36] and [T39]. We may not, however, infer the symmetric property *attached*$^{-1}$ (• *Live* •) ⊆ *Live*: as illustrated by the bold dark-green links on the last figure, there may be links from *Garbage* to *Live*.

The purpose of a GC is to remove all or some of *Garbage*. The inverse of [T35], *Objects ⊆ Live*, means that there is no garbage. It's not an invariant, but it will be the goal (the postcondition) of a full GC cycle. Let's give this property a name:

[D45] *No_garbage* ≜ *Live = Objects*

[T29], [T30] and [T36], and their fixpoint variants indicate that each of the sets *Objects*, *Heap* and *Live* is "stable" under *attached* and hence its closure, calling *A* stable under *r* if *r* (• *A* •) ⊆ *A*. The set *Stack*, on the other hand, is clearly not stable under *attached*; neither is *Garbage* because of the possible presence of links from *Garbage* to *Live* (the just noted dark-green links).

Invariants so far

As we will now consider events that may affect the state, and must prove that each of them preserves every invariant, it is useful to collect invariants seen so far:

[I3]	*Objects ⊆ Addresses*
[I6]	*attached* (• *Objects* •) ⊆ *Objects*
[I20]	*Stack ⊆ Objects –* **range** (*attached*)

Invariants

(*Final list on page 27.*)

2.3 OBJECT CREATION AND DESTRUCTION

Equipped with a model of the object structures, we now move on to a description of the the basic memory management operations that may affect them: object creation, stack allocation, and object deletion including full and incremental garbage collection.

All will be specified as B-like events, i.e. substitutions that occur under the control of a certain guard, with an Eiffel-like syntax.

> I use this syntax because I'm more comfortable with it, but don't be misled by appearances, this is not programming, it's mathematical specification in B style.

Object creation

Our first event models a basic instruction of an O-O language, of the form

> **create** *new*

new is a variable name, not a keyword.

(C++ syntax: *new_* = **new** *TYPE_OF_NEW* ();), which creates a new object and attaches it to *new*. Here is the corresponding mathematical event:

allocate (*existing*, *new*: *Addresses*) **is**
      -- Allocate a new object at *new*, chained to the object
      -- at *existing*.
  require   -- "PRE" in B
      from_live: *existing* $\in$ *Live*
      new_available: *new* $\notin$ *Live*
      new_virginal: *new* $\notin$ **domain** (*attached*)
  do   -- "THEN" in B
      Objects := *Objects* $\cup$ {*new*} ‖
      attached := *attached* $\cup$ {[*existing*, *new*]}
  ensure
      possibly_one_more: *Objects* = **old** *Objects* $\cup$ {*new*}
      linked: *attached* = **old** *attached* $\cup$ {[*existing*, *new*]}
      new_reachable: *Live* = **old** *Live* $\oplus$ {*new*}
      no_change_to_stack: **same** *Stack*
      no_new_garbage: *Garbage* = **old** *Garbage* – {*new*}
      rest_unchanged: **same** (*attached* \ (*Objects* – *existing*))
  end

The **do** clause contains two state changes, one affecting *Objects* and the other *attached*; they are separated by the symbol **||** to indicate parallel execution. (Another B notation, $x, y := e, f$ for multiple simultaneous assignments, would also work here.)

The postcondition uses the following two notations:

- **same** *x* is an abbreviation for $x = $ **old** *x*, expressing that the event doesn't change the value of *x*.

- "****" denotes restriction: $r \setminus A$, for a relation *r* and a subset *A* of its source set, is the relation made of all pairs $[i, j]$ in *r* whose first element *i* is in *A*.

So the clause *rest_unchanged* expresses that the event doesn't change anything in the relation *attached* except for the new link at *existing*.
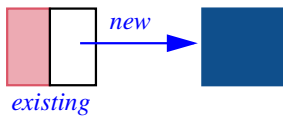
existing

Creation in O-O programing

The event's argument *existing* and the precondition clause *from_live* reflect the property that in object-oriented programming, as illustrated abovee, it is only possible to create an object from another object, which must be live: the instruction **create** *new* will be part of a class text, and will be executed on behalf of some instance of that class; if the object were not reachable, execution would have no way to get to that instruction. If *new* is an <u>attribute</u> of a class, *existing* is in the *Heap*; if *new* is a local variable of a routine, *existing* is an element of the *Stack*.

Also called data member (C++), instance variable (Smalltalk), field (.NET).

The event uses *new* as an argument, denoting the abstract address of the new object. In O-O languages you don't specify *new*, so we could remove this argument and instead use an integer outside of *Live*, chosen non-deterministically to satisfy the precondition. It is simpler to make *new* explicit.

The last two precondition clauses <u>state</u> that we may only reuse an address if is satisfies the following requirements:

- It doesn't host a live object (*new_available*).

- It's not in the domain of *attached*, that is to say, it doesn't contain a link to any object (*new_virginal*).

Let us now prove that the event *allocate* ensures its postcondition and preserves the invariants:

- The first two clauses of the postcondition, *possibly_one_more* and *linked*, simply restate the event's definition.

- For *new_reachable*: we are only adding one link, the pair [*existing*, *new*], to the relation *attached*, so any object in *Live* will remain in it. Adding the link implies adding *new* to *attached* (• *Live* •) and hence, from [T36], to *Live* itself. The second precondition clause, *new_available*, guarantees that this is a disjoint union. The only other way to add objects to *Live* would be through links from *new* to some other addresses (which would have had to be garbage or non-objects before the event); this is impossible because of the precondition clause *new_virginal*, without which we couldn't do this part of the proof.

- The clause *same_stack* is trivial since the event doesn't modify *Stack*.

- The clauses *same_garbage* and *rest_unchanged* follow from the preceding clauses.

- The invariant [I3] <u>stated</u> *Objects* ⊆ *Addresses*; its preservation follows from the postcondition clause *possibly_one_more* since *new* ∈ *Addresses*.

- The invariant [I6] <u>stated</u> *attached* ⊆ *Objects* ↔ *Objects*. The postcondition clause *linked* tells us that the event may add at most one element, *existing*, to its domain, *existing*, and at most one element, *new* to its range; this preserves the invariant since both will be members of *Objects*.

- The invariant [I20] <u>stated</u> *Stack* ⊆ *Objects* – **range** (*attached*) . Since the event does not change *Stack* and does not remove any element from *Objects*, it could only invalidate this property by adding a member of *Stack* to **range** (*attached*) . This means adding to *attached* a link leading into the stack. But the only new link leads to *new*, which from clause *new_available* is not in *Live*, and hence from [T34] not in *Stack*.

Getting the precondition right

The history of the two precondition clauses *new_available* and *new_virginal* is instructive. Initially I used a single clause *new* ∉ *Objects*, meaning that the allocation uses a memory cell at a fresh address. The correctness proof was straightforward. This form, however, is stronger than required: if we find that an object is in *Garbage*, it's OK to reuse its cell. So I replaced the clause *new* ∉ *Objects* by *new* ∉ *Live*, called *new_available* above. But then for the *new_reachable* postcondition clause, *Live* = **old** *Live* ⊕ {*new*}, I could not prove that the right-hand side is a subset of the left-hand side. The reason, it turns out, is that it is not always legitimate to recycle a garbage object *new*. If *new* has a link to another garbage object *go*, recycling *new* will make *go* — and possibly other objects as a result — reachable again, so the new *Live* will contain more than just **old** *Live* ⊕ {*new*}. Indeed we may only reuse a garbage object if it contains no links to other objects. The precondition clause *new_virginal* takes care of this, permitting the above proof.

This illustrates that even though the discussion relies on straightforward concepts t's still possible to make serious mistakes, which an attempt at mathematical proof will uncover.

> The consequences are not just theoretical. In a GC, failing to observe *new_virginal* is a real bug: failing to zero out a memory cell before recycling it. If the cell contains a link to a garbage object *go*, this will suddenly bring *go* back like Lazarus to the realm of the living. But a link to a live object *lo* — such as the dark green links in the earlier <u>figure</u> — is just as bad since *lo* will now have a spurious incoming link, preventing it from being reclaimed if, later on, all legitimate links to it disappear. Such a bug causes inexorably growing "memory leaks" and has plagued more than one released GC.

The theoretical difficulty goes away if we replace *new_available* and *new_virginal* by the stronger single clause *new* ∉ *Objects*. This would match the practice of O-O language implementations: it's fairly easy to keep track of the set of all *Objects*, whereas finding allocated objects that are not in *Live*, meaning they are in *Garbage*, usually requires performing a garbage collection; but then, having found the garbage objects, you might just as well remove them not just from *Garbage* but from *Objects* too. The event *collect_all*, introduced <u>next</u>, does this. Here we retain the weaker form of the precondition, since it is permissible, as we have just proved, to reuse any garbage object that you have been able to uncover, provided you zero it out to satisfy *new_virginal* and preserve [T7] (part of the invariant [I6]). This weaker precondition will become directly relevant when we consider incremental garbage collection, as modeled by <u>another event</u>, *collect_some*.

Full garbage collection

Although we'll need to model the <u>other form</u> of allocation (on the stack), for the moment we remain in the heap to examine events that *deallocate* objects, including both full GC and the freeing of individual garbage objects.

The following event represents a full garbage collection cycle

```
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
```

In the second assignment the replacement for *attached* is *attached \ Live*, denoting *attached* restricted to its pairs $[i, j]$ for which $i \in Live$; or, equivalently (see [T39]), *attached* deprived of all pairs for which $i \in Garbage$.

> The purpose of garbage collection is to restrict the set of objects to live ones, so the first assignment may seem sufficient; but if we didn't also restrict *attached* we might produce Big Brother elements in *Unused*, which keep references to objects without themselves being in *Objects*. Then the *allocate* event wouldn't be able to recycle such elements of *Unused* as objects, since they wouldn't satsify its precondition clause *new_virginal* as just discussed. It is precisely to avoid polluting the address space with such wasted elements that we introduced the property **domain** (*attached*) ⊆ *Objects* [T7] into the invariant [I6]; to preserve it, the event *collect_all* must update the relation *attached* in addition to the set *Objects*.

Let's prove that the event ensures the postcondition clauses and preserves the invariants. The first two clauses of the postcondition, *live_only* and *restricted_to_live*, restate the event's definition. The next one *restricted_to_kept* is a direct consequence of these two. The clause *no_change_to_stack* is trivial since the event doesn't affect *Stack*.

The clause *no_loss_of_life* states that the event *collect_all* does not change the set *Live*, defined [D33] as *attached** (• *Stack* •) , meaning *Stack* ∪ *attached* (• *Stack* •) ∪ *attached* (• *attached* (• *Stack* •) •) ∪ … Since the successive sets in this union are all in *Live* and the event doesn't change *attached* on *Live*, it follows that *Live* itself is not changed.

The clause *all_go_from_live* is an immediate consequence of *live_only*. For *all_lead_to_live*, assume a member *x* of **range** (*attached*) that is not in *Live*, and hence, because of *no_loss_of_life*, not in **old** *Live*. Relation *attached* must contain a pair of the form [*i*, *x*] for some *i* which, because of *all_go_from_live*, must be a member of *Live*, that is to say (again because of *no_loss_of_life*) **old** *Live*. Such a link from an object in **old** *Live* to an object not in **old** *Live* — a link from a live object to a garbage object, which must have existed before the event — is impossible as it would contradict [T36].

The property *all_live* follows from *live_only* and *no_loss_of_life*. The last clause *garbage_removed* is an immediate consequence.

If the invariant [I6], *attached* ⊆ *Objects* ↔ *Objects*, holds before the event, it will still hold afterwards as an immediate consequence of the postcondition clause *restricted_to_kept*. Note that this is only because we require the event, through its second assignment, to zero out all *attached* links in the garbage objects that it reclaims — making them, as noted, available for later recycling by the event *allocate*.

Since the event doesn't change *Stack*, it could invalidate the invariant [I20], *Stack* ⊆ *Objects* – **range** (*attached*), in one of only two ways:

- By removing from *Objects* a member of *Stack*; this is impossible because the postcondition clause *only_live_kept* tells us anything removed from *Objects* must have been outside of *Live*, and so, by [T34], outside of *Live*.

- By adding to **range** (*attached*) an element of *Stack*; this is impossible too since *live_only* tells us the event doesn't add any link to *attached* .

The reader, it is hoped, appreciates the eschatological significance of what has just been achieved. No later than page 17, we have managed to prove that if there is garbage it is all right to remove it.

A giant step for one man, not much of a step for mankind.

The free list

Practical garbage collectors do not always return the objects they collect to *Unused*, that is to say, to the operating system. In fact, only the best GCs achieve this; this means for example that on many versions of Unix they can't rely on the standard C routine *free*, which instead of releasing a cell from the memory of the current process simply adds it to a special data structure, the *free list*, from where it is available for reuse by the *same* process. Then no matter how many *free* operations you have put in your program, its process space will not shrink. Only if the GC uses special primitives such as the Unix *sbreak* will the process actually relinquish memory.

This observation suggests that to maintain the realism of our model we should include a variant of the *collect_all* event that, instead of moving garbage away from the set of *Objects* to *Unused* addresses, adds garbage objects to a special set *Free*. We'll have two new events: *free_all*, which adds garbage to *Free* without removing them from *Objects*; and *deallocate* which returns the elements of *Free* to *Unused*. Combining these two events in sequence must have the same effect as *collect_all*.

Two main advantages follow from this extension to our model:

•   We now have the flexibility of describing a GC that truly frees memory (*collect_all*), or can only return garbage to a free list (*free_all*), or returns garbage to the free list but occasionally deallocates the free list in *sbreak* style (*free_all* plus *deallocate*).

•   We can now model *incremental* garbage collection. As will be seen in the next section, this would be impossible without the notion of a free list, because returning a subset of *Garbage* to *Unused* would violate the No Big Brother property. The introduction of *Free* permits a variant of *free_all* that collects some but not all of the garbage.

We introduce the free list as a variable:

> [A46] *Free*: $\mathbb{F}$ ($\mathbb{N}$)   ●

Elements of *Free* must be objects (otherwise we can't have links to them without violating No Big Brother); the only place where we can meaningfully have them is garbage, hence an invariant:

> [I47] *Free* $\subseteq$ *Garbage*   ●

In addition, elements of *Free* must be immediately reusable for the allocation of new objects — that's their whole raison d'être —, so they must satisfy the precondition clause *new_virginal* of the *allocate* event by not having any outgoing links, a condition we express through another invariant:

> [I48]   *Free* $\cap$ **domain** (*attached*) = $\varnothing$

In practive this means that any event that adds elements to *Free* must remove their outgoing links. The following figure illustrates the situation: it shows the set *Garbage*, extracted from the earlier picture of the overall object structure, with its new subset *Free*:

Garbage and free list
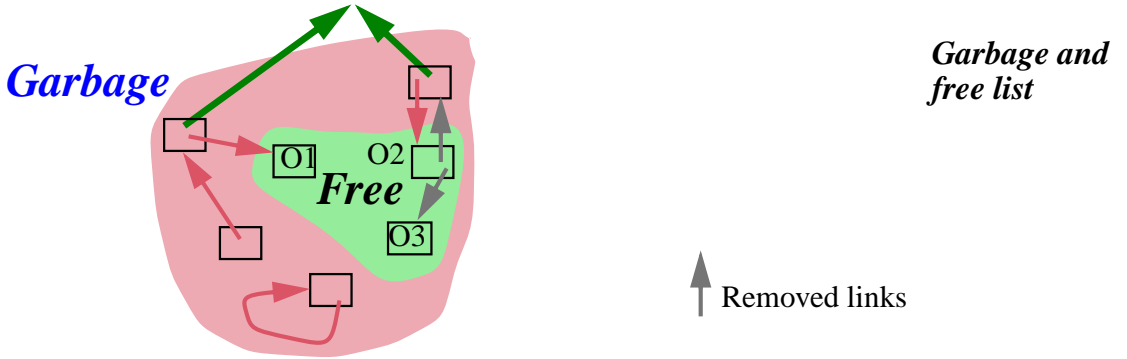
As per [I48], all *outgoing* links have been removed from the objects in *Free*; but there may still be *incoming* links into these objects, such as the red links into O1 and O2. Such links may only come from non-free garbage as per the following theorem:

> [T49]   *attached*$^{-1}$ ($\cdot$ *Free* $\cdot$) $\subseteq$ *Garbage* – *Free*

Proof: incoming links into *Free* may not come from *Free* because of [I48]; they may only come from *Garbage* because of [T44]. We may in fact infer from this a stronger property (although [T49] will be the useful form):

> [T50]   *attached*$^{-1}$ ($\cdot$ *Garbage* $\cdot$) $\subseteq$ *Garbage* – *Free*

Since the previous events did not involve *Free*, they all preserve our two new invariants.

The new event *free_all* — which will also preserve them — is almost identical to *collect_all* (the specification below highlights the differences) but returns freed elements to *Free* rather than *Unused*:

```
free_all is
        -- Move all garbage objects to the free list.
    do
        Free := Garbage
        attached := attached \ Live
    ensure
        live_and_free_only: Objects – Free = old Live
        restricted_to_live: attached = old (attached \ Live)
        restricted_to_kept:
                attached = (old attached) \ (Objects – Free)
        no_change_to_stack: same Stack
        no_loss_of_life: same Live
        all_from_live: domain (attached) ⊆ Live
        all_to_live_or_free: range (attached) ⊆ Live ∪ Free
        all_live_or_free: Objects = Live ∪ Free
        garbage_freed: Garbage = Free
    end
```

Proofs of the postcondition and previous invariants are the same as for *collect_all*; proofs for the two new invariants are immediate.

As illustrated on the last figure, the state resulting from *free_all* may still have *incoming* links into *Free*.

The following event removes any such links by deallocating the free list:

```
deallocate is
        -- Get rid of all objects in the free list.
    require
        recyclable: attached⁻¹ (• Free •) ⊆ Free
    do
        Objects := Objects – Free    ‖
        attached := attached \ Free    ‖
        Free := ∅
    ensure
        rid_of_free: Objects = old Objects – Free
        -- Other clauses omitted (see previous events)
    end
```

The precondition requires that incoming links into *Free* come only from *Free* itself, rather than from non-free *Garbage* elements (see [T49]). This is not the case for example on the figure of page 19 unless the two links going to O1 and O2 are taken away. Without this, *Objects := Objects – Free* would create Zombie links from *Objects* to *Unused*; even if such links can only originate from *Garbage*, they stilll violate the Basic Object Constraint.

If, on the other hand, no links exist to *Free* objects except from *Free* itself, then the precondition of *deallocate* is satisfied. This is the case in particular when all garbage has been made free:

> (*Free = Garbage*) ⇒ *recyclable*
>   -- Here *recyclable* is the precondition of *deallocate*

(Proof: follows directly from [T44].) This means that it is legitimate to use *deallocate* just after *free_all*, thanks to the latter event's last postcondition clause, *Garbage = Free*.

Incremental garbage collection

The events *collect_all* and *free_all* represent a full GC cycle that removes all garbage. In a modern language implementation there must also be room for an incremental GC, which removes some garbage objects but not necessarily all.

This suggests that we need another event *collect_some* (*Rejects*) whose argument *Rejects* denotes a set of garbage objects. As a special case, *collect_some* (*Garbage*) will describe the same operation as *free_all*. At the other extreme, we can use *collect_some* ({*o*}) to describe the collection of a single object *o*, or , in a non-GC-language, a programmer-controlled operation to free this object safely.

Here is *collect_some* in a form as close as possible to *free_all*:

collect_some (*Rejects*: $\mathbb{P}$ (*Objects*)) **is**
      -- Get rid of all the objects in *Rejects*.
  require
      recyclable: *Rejects* ⊆ *Garbage*
  do
      Free := *Free* ∪ *Rejects*
      attached := *attached* \ $\overline{Rejects}$
  ensure
      rejects_freed: *Free* = **old** *Free* ∪ *Rejects*
      restricted: *attached* = **old** (*attached* \ $\overline{Rejects}$)
      restricted_to_kept:
          attached = (**old** *attached*) \ (*Objects* – *Free*)
      no_change_to_stack: **same** *Stack*
      no_loss_of_life: **same** *Live*
      from_live_or_other_garbage:
          domain (*attached*) ⊆ *Live* ∪ (*Garbage* – *Rejects*)
      -- No counterpart to *all_to_live_or_free*
      all_live_or_free_or_other_garbage:
          Objects = *Live* ∪ *Free* ∪ (*Garbage* – *Rejects*)
      no_change_to_garbage: *Garbage* = **old** *Garbage*

      -- Following have no counterpart in previous events:
      no_change_to_objects: *Objects* = **old** *Objects*
      possibly_more_free: **old** *Free* ⊆ *Free*
  end

The replacement for *attached* is *attached* \ $\overline{Rejects}$, meaning: *attached* deprived of all pairs of the form [*i*, *j*] where *i* ∈ *Rejects*. The proofs are similar to those of the last two events and left to the reader; the key property is the precondition *recyclable*, without which we couldn't guarantee *no_loss_of_life*.

All clauses except the last two are counterparts of those of *free_all*, the same or weaker. The reason for using *free_all* as our model, rather than *collect_all*, is that *collect_some* must be applicable to an arbitrary subset *Rejects* of *Garbage*: then it cannot remove the corresponding objects from *Objects*, like *collect_all*, since as already noted any links into *Rejects* from the outside (which can only come from other *Garbage*) would yield Zombie links. So here the only possibility is a *free_all*-like behavior that moves the *Rejects* to the *Free* lists without removing them from the set of objects.

Removing the objects for good — sending them to *Unused* — means having a *deallocate* take place after *collect_some*, which is only possible if *attached*⁻¹ (• *Rejects* •) ⊆ *Rejects* to ensure the precondition of *deallocate*. (This is the case if *Rejects* is all of *Garbage*; indeed *collect_some* (*Garbage*) is the same as *free_all*.) Of course we could add this clause to the precondition of *collect_some* itself, enabling this event to remove the *Rejects* completely; but then the model ceases to be realistic since an incremental collection cycle would now need to find all the links into *Rejects* and hence to work on *Garbage* as a whole, whereas the very notion of *incremental* garbage collection implies that if you have spotted a few *Rejects* you can free them without having to traverse the rest of the *Garbage*. So the best *collect_some* can do in the general case is to move the *Rejects* to the *Free* list.

This reasoning is one of the principal justifications for introducing the notion of free list into the model.

Recycling an object

Even if prevented to return its *Rejects* to *Unused*, an incremental GC cycle will remove them from **domain** (*attached*). That's enough to make them available to the event *allocate*, which needs a suitable address for a new object. As we have seen, that doesn't mean an address outside of *Objects*, just outside of both *Live* and (clause *new_virginal*) **domain** (*attached*). The postcondition clause *from_live_or_other_garbage* of *collect_some* ensures it.

As a consequence we may define an event that, by combining *collect_some* and *allocate*, frees an object and immediately reuses its address for a new object:

```
recycle (existing: Live; reject: Objects) is
        -- Reuse the address of new to allocate a new object,
        -- chained to the object at existing.
    require
        from_live: existing ∈ Live
        new_recyclable: reject ∈ Garbage
    do                   -- Two events in sequence:
        collect_some ({new})
        allocate (existing, new)
    ensure
        … Left to reader (see postconditions of events
            collect_some and allocate) …
    end
```

Its correctness requires that, as just noted, the precondition of *allocate* hold after *collect_some*. The postcondition and the rest of the proof are left to the reader.

Stack allocation

The previous events had to do with objects allocated on the heap. In the execution of an object-oriented we also need a stack-based form of allocation, similar to pre-O-O techniques as present in Algol 60. The following event provides it; routine calls will use it for every local variable and by-value argument of a reference (non-expanded) type.

> *allocate_on_stack* (*new*: *Addresses*) **is**
>   -- Allocate a new stack object at *new*.
>   **require**
>   *new_available*: *new* $\notin$ *Objects*
>   **do**
>   *Stack* := *Stack* $\cup$ {*new*}
>   **ensure**
>   *root_added*: *Stack* = **old** *Stack* $\oplus$ {*new*}
>   **end**

Proving the postcondition and the preservation of the invariants is easy. Note that here we cannot any more weaken the precondition *new_available* to *new* $\notin$ *Live* **and** *new* $\notin$ **domain** (*attached*) as <u>we did</u> for the event *allocate*: if *new* is a garbage object, other garbage objects may have links to it; then if we attempted to recycle it as a stack object we couldn't any more guarantee the postcondition, which requires *new* to be part of the stack and hence, from [T21], to admit no incoming link. So for the choice of *new* we exclude all currently allocated addresses, live as well as garbage. This matches the behavior of practical memory allocation schemes, which draw stack addresses and heap addresses from different address pools.

One may similarly define an event *free_from_stack* (*existing*) that removes an element from the stack. This is left to the reader.

2.4 REPRESENTING ADDRESSES

The set of object addresses was specified as a set of integers:

[A1] *Addresses*: $\mathbb{F}$ ($\mathbb{N}$)

Do we indeed need to know what *Addresses* is? Not at this stage; we could proceed for a while without stating what *Addresses* is made of. One might even conjecture that this choice of a concrete set for *Addresses* betrays that the author of this discussion is a programmer, faithful to the usual mores of his species: implement first and maybe think later.

The choice is indeed the mathematical counterpart of what in software would be an implementation decision. But it seems justified if the goal is to build a useful model of the execution of programs on computers. Our computers have memories, and these memories have sequentially numbered cells. Hence the idea of defining *Addresses* as a set of integers. Even if early on we don't care about this aspect, it becomes relevant if our eventual aim is to formalize O-O programs, or even just their GCs.

How much low-level an implementation decision this is depends on the intuitive semantics we attach to $\mathbb{N}$. Fortunately we can continue developing the model without choosing our exact level of abstraction:

• Under a "*high-level*" interpretation we may think of an abstract memory, where each cell (denoted by an integer from the domain of *attached*) contains an object — an instance of a class. For example the GC of versions 1 and 2 of ISE Eiffel relied on having all live objects linked together (through a hidden field added to every object); it also chained together in a "free list" all the dead objects it reclaimed. With such an interpretation every integer denotes not a physical address in the computer's memory but a position in a list of objects.

• We may also use a "*low-level*" interpretation and consider the integers to be the actual starting addresses of the objects' representation. One of the pleasant consequences of using relations — or, starting with the next article, possibly partial *functions* — is that they don't have to be total or surjective. So if an address doesn't correspond to the beginning of an actual object (in particular, if it is not a multiple of 4, assuming objects start on 32-bit word boundaries and our addresses are counted in bytes) it will simply not be in *Objects* and hence will not denote any object.

This interpretation also yields (when combined with the Basic Object Constraint) an important property of the model: *no references to subobjects*. In Eiffel, even though the framework supports subobjects (through the "expanded" mechanism), a reference will always be attached to a first-level object, never to a subobject. The experience of early language and compiler versions showed that permitting references to subobjects complicates the GC and precludes some optimizations, for no significant expressive benefit.

With the low-level interpretation we will most likely go further in our specification of *Addresses* and define it as the interval $1\,.\,.memory\_high$ for some non-negative integer *memory_high*. From there one can start discussing in detail the properties of a memory management scheme. For example a GC of the *mark-and-sweep* kind needs in its "sweep" phase to traverse the whole memory — the interval $1\,.\,.memory\_high$.

This interpretation may also allow us to take into account the actual content of objects (their "expanded" fields), not just — as in the present discussion — the references they contain to other objects. The content of an object identified by the integer *n* (assuming *n* belongs to *Objects*) is simply what's stored at a set of physical addresses starting at *n* and bounded by *next* (*n*) – *1* where *next* (*n*) is the next member of *Objects*, if any.

At this stage, however, nothing forces us to disallow subobject references, or to choose the "high-level" interpretation or the "low-level" one, or any other. Their role is simply to reassure ourselves that the mathematical model is realistic.

What has been postulated

Here for convenience is a recapitulation of the assumptions made so far (marked originally with ● signs): seven axioms, each postulating an element of a known set, and seven invariants postulating properties of these elements. They make up the basis of what we need to reason about run-time object structures. Every one of the events to be studied now will have to preserve the invariants.

[A1]	*Addresses*: $\mathbb{F}$ ($\mathbb{N}$)	
[A2]	*Objects*: $\mathbb{F}$ ($\mathbb{N}$)	
[A5]	*attached*: $\mathbb{N} \leftrightarrow \mathbb{N}$	
[A19]	*Stack*: $\mathbb{P}$ (*Addresses*)	
[A46]	*Free*: $\mathbb{F}$ ($\mathbb{N}$)	

Axioms

[I3]	*Objects* $\subseteq$ *Addresses*
[I6]	*attached* (• *Objects* •) $\subseteq$ *Objects*
[I20]	*Stack* $\subseteq$ *Objects* – **range** (*attached*)
[I47]	*Free* $\subseteq$ *Garbage*
[I48]	*Free* $\cap$ **domain** (*attached*) = $\varnothing$

Invariants