Explicit Deallocation without Dangling References II

ANDRZEJ SALWICKI, Cardinal Stefan Wyszyński University
ANDRZEJ ZADROŻNY, Institute of Computer Science PAS

This paper appears 30 years after the article by Gianna Cioni and Antoni Kreczmar under the same title. In the original paper [Kreczmar 1987], Kreczmar introduced a memory management system for dynamic allocation of objects with a few interesing properties. While leaving the programmer in control of object dealloacation (which in Kreczmars system is explicit), the system is free of the dangling reference problem, i.e., following the explicit deallocation of an object, all references to the object within the program become automatically nullified. In the present paper, we compare the memory management system described in [Cioni and Kreczmar 1984] with the solutions that appeared later (cf. Table I. below). We also give a formal specification for Kreczmars allocator in the form of a formalized algorithmic theory ATHM and sketch the construction of a programming model of the specification. Thus we prove that the specification is consistent, i.e., free of contradictions, and hence it is feasible.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Memory management (object deallocation)*

General Terms: Design, Performance, Reliability

Additional Key Words and Phrases: dangling reference, garbage collection, objects, programming languages, safety

ACM Reference Format:
ACM Trans. Program. Lang. Syst. V, N, Article A (January YYYY), 17 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

The management of objects, especially the problem of efficient and safe memory allocation, plays an essential role in object-oriented programming. A practical system addressing the problem must be safe (i.e., a deallocated object should be clearly perceived as absent from all places in the program, preventing an unintended interpretation of "bad memory" as the contents of an object that formally no longer exists) and efficient, meaning that the cost of referencing an object (dereferencing an object pointer) cannot increase too significantly beyond the cost of a simple (machine-level) indirect memory reference. Typically, the dynamic instances of a program's modules reside in two disjoint areas in memory: the stack and the heap. The activation records of functions, methods, contructors, and blocks are allocated on the stack, while the dynamic instances of structures, classes, and, sometimes, more exotic objects, like coroutines (for those languages that admit such objects) are stored in the heap. That latter category of objects is of our primary concern. Owing to the fact that, unlike the former category, the patterns of their creation and destruction do not follow a simple ordered paradigm,

Author's addresses: Andrzej Salwicki, Department of Mathematics and Natural Sciences, Wóycickiego 1/3, 01-938 Warsaw, Poland, mailto:salwicki@mimuw.edu.pl; Andrzej Zadrożny, Institute of Computer Science Polish Academy of Sciences, 5 Jana Kazimierza Str.,01-248 Warsaw, Poland

DOI: http://dx.doi.org/10.1145/0000000.0000000

but are essentially random and arbitrary, the mechanisms of their allocation and requisite bookkeeping are open for discussion and invention, being thus also susceptible to trends. The first generation of popular programming languages with rudiments of objects: Pascal and C, opted for simplicity (read efficiency) at the cost of safety. Following the (explicit) deallocation of an object, any replicate references to the object within the program would still point to the old chunk of memory originally used to store the object, thus creating a hazard. It was up to the programmer to sidestep that hazard by not touching the old references (pointers) until reset. Note that a mistake consisting in a reference to a deallocatad object would not, in those circumstances, translate into an immediate error: a read access would return an incorrect result (that might trigger an error a potentially long while later), while a write access would potentially corrupt some legitimate object, often with delayed symptoms difficult to diagnose. When C++ appeared later, inheriting and absorbing the legacy of C under its umbrella, it did not revolutionize the paradigm of memory allocation for objects, primarily for two reasons: (1) C++ still preferred efficiency over safety (it was supposed to replace C without affecting the implementation aspects of its C subset) (2) C++ wanted to be compatible with C (C functions, using the traditional memory allocator, had to be able to coexist with C++ functions within the same program allowing both types of functions to operate on the same pointers) Java brought a response to the lack of safety inherent in the traditional approach to memory allocation for objects in the postulate to completely hide the action of object deallocation from the programmer's view. By abandoning pointers (plus the large bag of pointer trick available to experts in C and C++), and replacing deallocation with the implicit and invisible garbage collection, Java also simplified object-oriented programming, thus making it accessible to a wider population of programmers (who no longer needed to be experts on such low-level and mundane aspects of computing as pointers). C#, introduced by Microsoft as their own "Java", was basically the same creation when seen from the viewpoint of run-time memory management. That global paradigm shift was compatible with the rapid advancements in CPU and RAM technologies making large and sloppy (less efficient) programs vastly more acceptable than in the early days of C++. These days, few people care whether the efficiency of non-number-crunching program can be improved by the factor of 10 or 20 in terms of execution speed or memory demands for as long as the program appears to work. Following the short initial euphoria, the sealed, one-size-fits-all memory allocation system of Java (and friends) begun to exhibit its own shortcomings. The problem was the inability of the programmer to indicate to the garbage collector that some objects were less needed than others, even though references to them were still present in the program. The standard example of a situation where this kind of indication would be useful is opportunistic caching where, by definition, an object stored in the cache (and thus being referenced by a "pointer") can (should) be removed when memory becomes scarce. Thus, in 1995 (i.e., three years after the introduction of the first version of Java), the concept of weak reference was added to the language. The idea was to let the garbage collector remove an object, if it was reachable solely via weak references. Notably, that solution didn't quite solve all problems. As of today, there are four reference types in Java: strong, soft, weak, and phantom (in the decreasing order of strength) whose role is to cater to the various flavors of discarding an object without eliminating its reference first. Judging by the amount of discussion on the web, the four reference types cause more problems to apprentice Java programmers than pointers ever did to newcomers to C or C++. In the latter case the issue was quite simple: one either understood pointers in their entirety or not at all, whereas in Java the situation is exacerbated by the fact that one can go some way without any understanding of the underlying problem at all. The problem, of course, is that there are important circumstances where the cog- nizant programmer would (should) prefer to exercise more or

less direct control over memory allocation for objects. That was realised quite promptly, also within the community of (expert) Java developers, despite their initial optimistic attitude towards garbage collection as the sole and ultimate remedy for all problems of memory allocation. In this context, as the pendulum seems to have reached the end of its swing, and is in fact beginning to move in the opposite direction, we believe that the time is ripe to revisit the valuable and somewhat forgotten ideas behind a solution that strikes a compromise between the two extremes, i.e., direct (unsafe) deallocation and (total, unqualified) garbage collection. This solution comes as safe programmed deallocation.

2. MANAGING OBJECTS

Proper management of objects plays an essential role in execution of object-oriented programs. It must ensure safety and efficiency. Poorly designed ways to remove objects appear unsafe. On the other hand, keeping unnecessary objects slows down the calculations and may lead to a collapse.

It is commonly accepted that dynamic instances of program's modules reside in two non-overlapping fields in memory. The activation records of procedures, functions, constructors, and blocks are alocated on a stack. The dynamic instances of classes (and also of coroutines, if a programming language allows) are alocated on a heap. This paper presents a choice of problems related to the management of heap. In 1980 Antoni Kreczmar conceived a running system (a virtual machine) for Loglan programming language c.f.[Kreczmar 1987]. Later, in the paper [Cioni and Kreczmar 1984] an the object management system which is free of dangling references and is efficient has been described. Hanna Oktaba studied the system and constructed the algorithmic theory of references and analysed its metamathematical properties [Oktaba 1982]. The papers of Cioni and Kreczmar and of Oktaba accomplished the task of *verification* of the Kreczmar's system. The system has been thoroughly *validated*, for it is in use since more than 30 years as a part of the running system of Loglan'82 object programming language [Salwicki 2013].

We believe the time has come to compare the system of Kreczmar to other object management systems and to explain it in terms of an algorithmic theory (different from the Oktaba's theory).

Most of presently used object oriented programming languages appeared after 1984 (C++ in 1985, Java in 1995, Python in 1989, Ruby 1993 [Gupta A. 2010]). No one of them has a system similar to the Kreczmar's one.

In C++ deallocation of objects is fast. However, it is well known that the system of C++ has no protection against dangling reference errors. The instruction `delete()` is executed without any form of control. The same may be said about the programming language Pascal and its instruction: `dispose()`.

Java programming language forbids the explicit removal of objects. Its system is free of dangling reference errors, however it is inefficient.

We think that it is worthwhile to discover the inventions contained in [Cioni and Kreczmar 1984] and to apply them in practice, for they offer the safety without any loss on efficiency.

Let us briefly expose the problems. Objects are 1°created, 2°shared, 3°used, and, 4°eventually become not needed, anymore. We shall abstract from many technical details of object's creation such as its size, the type of object, the ways the free memory is organized.

Creation of objects is done through evaluation of object generator expressions, e.g. in the assignment `x := new ClassType(actparams)`. An object, once created, can be shared among several variables, inspected, and updated.

Sharing is accomplished by execution of assignment instructions e.g. the assignment `y:=x` causes that now the object pointed by x is shared by the variables x and y.

Other forms of object's usage reduce to one of three cases:
inspection – reading the value of an attribute `attr` of an object e.g. `x.attr` ,
updating – writing the value of an attribute, e.g. `x.attr:=8`,
servicing – done by calling a method of the object e.g. `call x.meth(81)`.
All three forms of usage should begin with the checking whether the variable `x` points to an alive object.

Eventually no further actions on the object will be taken. It is wise to dispose of it. In these circumstances the designer of an object management system is confronted to three major threats:

— *memory leak* problem,
— memory *fragmentation*,
— *dangling references* problem.

Below, we explain these terms:

memory leak	occurs when objects are created and remain unused. Program consumes memory. It leads to the slowdown of computations or even to a complete blockade.
dangling reference	A situation when in past some variable x pointed to an object o, but at present, the object o no longer exists. One says, the variable x is a dangling pointer if any attempt to use the variable x is treated **without** an alarm. The system is free of dangling reference error if any attempt to use the variable x raises an exception, e.g. caused alarm: `reference to none`.
contradiction	Two variables of different types mutually contradict themselves: variable x says *I am pointing to an object of type A*, variable y says *I am pointing to an object of type B* and both variables point to the same address. This situation may happen when deallocation created dangling reference(s).
destruction	Suppose the object memory system admits the dangling references. It may happen that, after execution of `delete(x)`, the dangling pointer y points to the memory frame where a new object z resides. Then the delayed instruction `delete(y)` will cause the destruction of the object z.
fragmentation	There are many slots of free memory, none is large enough to hold a new object.

A few words on dangling reference error are in order: one may distinguish between detected and undetected dangling reference error. The second error is a real danger. Detected dangling reference happens when the virtual machine finds that the value of a pointer is `none` (or `null`). This is an unpleasant situation, for the programmer must find the cause of it. Undetected dangling reference is much worse for many reasons, see above. One may ask: *if so then perhaps it is possible to equip the compiler in a tool to detect dangling references errors in advance – before an execution of a program?* The answer is: NO, such an algorithm does not exist. [1]

There exists a variety of object management systems. One may classify them with respect to different definitions of garbage.

[1] Should an algorithm $\mathcal{A}$ for detection of dangling reference errors in source program exist, then we would construct another algorithm $\mathcal{B}$ to detect whether any given program will terminate or not. This is known to be impossible.

The first, most natural, definition of garbage reads "*An object o is a garbage, whenever the program instructs* (the runtime system) *it is no longer needed.*". This definition is accepted in C++ programming language. A C++ program may contain instruction `delete(x)`. However, the instruction has a side effect: the *dangling reference* may appear. Namely, one can observe some variables that point to segments of memory where no object resides. A dangling reference may lead to another error of *contradicting information*. This phenomenon occurs when two variables point to the same address and one says: *"I am pointing to an object of type A"* and the other says *"I am pointing to an object of type B"*. Errors of both kinds are difficult in diagnosis and very dangerous ones. There is no algorithm to detect the dangling reference in the text of program. For the problem is reducible to the halting problem.

Another definition of garbage reads: "*Object with no references to it, is a garbage*". Some programming languages try to keep track of references with reference counters (e.g. Python [Wikipedia 2013b]). In this way a garbage collector can easily identify objects with reference counters equal zero as garbage. However, by introducing reference counters one creates an overhead in code's length and also in execution time. The result is a slowdown of execution (A. Appel says "On the whole, the problems with reference counting outweigh its advantages",[Appel 1998] p.264). The same opinion had O.-J. Dahl, the father of object oriented programming [Dahl 1974]. Let us recall that reference counters do not help in recognition of cycles of no longer needed objects.

Subsequent definitions of garbage base on different types of references. Namely, one differentiates *weak* references from normal ones. Now, "*Object o is an garbage if there is no normal reference to it, even if there exist weak reference to o*". The weak references were introduced with two aims: 1° to decrease the cost of reference counting, 2° to diminish the risk that some objects will be kept because the programmer forgot to nullify all references to it.

All three types of object management systems have certain deficiencies. The question arises: is it possible to replace operation delete by another operation, say kill, such that kill has no side effect of dangling reference. In [Cioni and Kreczmar 1984] it was shown that there exists an objects management system with kill operation.

Any program that intensively creates objects and deallocates some, may cause *fragmentation* of the object memory. Sometimes, the situation may be improved by the defragmentation.[2]

The system designed by Kreczmar integrates the features mentioned above. For it allows to:

— deallocate no longer needed objects (kill operation),
— compactify heap of object (defragmentation),
— collect garbage, i.e. objects that are not accessible.

Below we compare object programming languages. We indicate that it is desirable that the languages and running systems satisfy the following conditions:

r1) For every type (class) T, for every variable x. If the variable x is of type T then its value is an object of a subclass U of class T or $x = none$.
(This is a fundamental invariant (i.e. axiom) of any true object system)
r2) For every object s it is possible to distinguish the fields containing pointers to objects from the fields that hold values of primitive types, like e.g. integer, float, boolean,...

[2]In [Cioni and Kreczmar 1984] the authors use the word compactification.

3. DEALLOCATION IN JAVA, C++, PYTHON ET AL.

In this section we briefly present the solutions taken in other object oriented languages.

3.1. C++, object Pascal, Objective C

We shall limit our considerations to the programs free of *malloc* instruction. For malloc instructions break the rules r1, r2, listed above.

Object y can be deleted with $delete(y)$ statement. The effect of this instruction can be exemplified by the following implication

$$\underbrace{(x == y == z! = null)}_{precondition} \implies \underbrace{\{delete(y); y = null;\}}_{\text{statement}} \underbrace{(y == null \wedge x == z! = null)}_{postcondition}.$$

The variables x, z that were pointing to the removed object preserve their value. It leads to the dangling references error. It is normal and expected that an attempt to read the value of $y.attr$ throws an exception. However an evaluation of $x.attr$ may return a nonsensical value instead of exception – this is the danger of dangling reference error – a signal is not raised when it should be.

The error can be avoided if all those variables are nullified $x = null; ...z = null;$. It is a task of a programmer to remember all variables referring to the object getting deallocated and to nullify **all** of them prior to instruction delete. Obviously it is an error prone approach. An automatic completion of the instruction `delete(y)` by the instruction `y=null` can be done easily in many ways. However, it is programmer only who can add the instructions `x=null; z=null;`

3.2. Java, Python et al.

Already the report on language Modula3 [Cardelli et al. 1989] drew attention to the risk of hanging references and non-existence of an algorithm that could detect such errors. In Java white paper [Gosling and McGilton 1995] there are quite a few statements describing this problem and justifying the interdict of $delete$ instruction. Instead of $delete()$ instruction there is a **Garbage Collector** mechanism, which frees programmer from manual removing of references to objects.

Soon, the opinion that garbage collector is an ultimate solution in objects management was verified. Three years after first version of Java (1995), in JDK 1.2, weak references have been introduced. It turned out, that in many cases a programmer forgot to nullify all references to an object destined to deletion, or one simply was not aware of some references created by a data structure. To decrease the number of such errors the notion of weak reference was proposed. [Wikipedia 2013c]. A developer can declare variable as weak reference. Weak references do not change reference counter in cPython [Wikipedia 2013a], and garbage collection algorithm does not take them into account in Java. If there are no strong references to object it is considered for collection. Even if there are some weak references to it.

Remark 3.1. Let's consider the following situation from Java:

$$x_1 : x_1 \to o \quad \text{and} \quad y_{weak} \overset{weak}{:=} x_1$$

One may say: if weak reference y refers to some object o, then there exists a strong reference x_i to the same object

$$y_{weak} : y_{weak} \to o \Leftrightarrow (\exists\, x_i)\, x_i \to o$$

This is only partially true. After operation $x_1 := null$ the weak reference y_{weak} continues to refer to original object o for some time. Only after run of garbage collection mechanism object is disposed, and weak reference is nullified. Due to non-deterministic implementations of most garbage collectors developers cannot predict, nor effectively enforce collection.

□

Remark 3.2. As long as weak reference to the object exists one can create a situation when object intended for collection will be restored to life:

```
Disposable tg = new Disposable();
   /* A new Disposable object o is created, tg is a reference to the object o */
WeakReference<Disposable> weak_tg = new WeakReference<Disposable>(tg);
   /* creates Weak Reference to the same object o */
tg = null;     /* remove last normal reference to the object o
   The object o is ready to be collected */
System.gc();     /* This is a hint – not an obligation – to activate GC */
Disposable tg2 = weak_tg.get();
   /* It may happen that GC was not activated.
   And operation get  will reestablish a strong reference to the object o */
```

This will happen when instruction $System.gc()$ will be ignored for some reason by the virtual machine.

□

3.3. Tombstones

There is a way to handle objects that allows to deallocate objects and to avoid dangling references. The technique is known as tombstones cf. [Lomet 1975], [Gabbrielli and Martini 2010] p.248. Every time an object o is created (e.g. by execution of `x:= new C(...)`), the virtual machine creates an additional object t - the tombstone of the object o. The content of the tombstone is either physical address of object o or null. The value of the variable x of type C is the physical address of the tombstone t. An assignment `y:=x;` copies the address of t to y. Any access to the object o requires two memory cycles. Deletion of the object o can be done safely. It suffices to put null as the new value of the tombstone t and recycle the memory frame occupied by the object o. It seems that no object programming language uses this technique. There is some amount of prejudice concerning the cost of tombstones. Most comments repeats that the overhead is too big. One may observe that these comments are not accompanied by any form of arguments. We may add that the extra cost in time and space is worth its price for the threat of dangling references is enormous. Moreover, the critique of costs was written some 30 years ago when the speed of computers and the size of memory were significantly smaller. Tombstones seem to be an ideal solution. For they offer the safety for a reasonable price. However the tombstones have some drawbacks. Namely, there is no way to get rid of tombstones. They accumulate and we are confronted with the phenomenon of memory leak again.

An attempt to overcome this problem was proposed in [Fisher and Leblanc 1980]. We quote the entire description of their proposal.

"An attractive method of pointer checking is to represent a pointer as a pair (key, address) and head each allocation from the heap with a lock field. When a pointer is used, the key value must agree with the lock field of the object referenced. This is again a very efficient run-time test. It does not provide absolute security since the key field is simply a bit pattern that could be fabricated by a malicious user. This relative security is acceptable if accidental fabrication is very improbable.".

The structure is known as "keys and locks" c.f.[Gabbrielli and Martini 2010]. Kreczmar, independently found a similar solution. The paper [Cioni and Kreczmar 1984] contains details of implementation as well as the proof of correctness.

3.4. Comparison

In the Table I. we compare the ways different programming languages deallocate objects. We distinguish three groups of the languages: the first group consists of the Loglan'82. The languages of the second group admit programmed deallocation (C++, Pascal, etc.) The third group contains the languages that forbid deallocation, and rely on garbage collection. The following five aspects were taken into consideration: **Pre**-condition - common in all cases, **Code** - that leads to deallocation, **Post**-conditions - observe the differences, **Cost** - order of time units needed, **Risk** - that deallocation fails and leaves the object intact, or that an error occurs.

Some explanations concerning cost of deallocation operation seem necessary: The cost of `delete()` in C++, `dispose()` in Pascal, and `free()` in Ada are known [Stroustrup 2013; Jensen and Wirth 1974; Barnes 1996]. The cost of `kill()` is calculated in [Cioni and Kreczmar 1984], and it will be explained below, see Appendix A. The cost of any garbage collector `gc()` is known as $O(m)$, where m is the size of heap i.e. object memory. Note, any garbage collecting algorithm must visit each object in the heap.

Table I. Dealocation of objects in various programming languages.

	Loglan'82	C++, Pascal	Java, Python
Pre-	Certain object o is referenced by the variables $x_1 = x_2 = \ ... \ = x_n, \quad 1 \leq i \leq n$.		
Code	kill(x_i)	delete(x_i); $x_i = null$	$x_1 = null$; $x_2 = null$; ... $x_n = null$; Now, the instruction $gc()$ the object o will be deleted.
Post-	All the variables took the value **none**. Object o is deleted.	Object o has been deleted. The variable x_i has the value null. Other variables point to the deleted frame – it is a *dangerous error* – dangling reference.	Object o has been deleted – under condition that all the strong (normal) references to the object have been earlier assigned the null value.
Cost	$O(1)$	$O(1)$	$O(n+m)$ m is the global size of the heap of objects.
Risk	No risk(!) For each attempt to read and/or write from the deleted object will raise an error signal {reference to none}.	If n>1 then dangling reference error occurs. High probability of the error of contradicting information and/or destruction error.	Chances that programmer will forget to nullify some reference to the object o and hence that the object will remain not deleted.

4. ALGORITHMIC SPECIFICATION OF KRECZMAR'S SYSTEM

Below, we present a specification of the Kreczmar's heap management system. A specification S is an extension of an interface by a set Ax of algorithmic formulas. The set Ax may be used in the process of analysis of an application. We present an example of reasoning on a program in Appendix B. The formulas of the set Ax are used as *axioms*. The same set of formulas may be used as a criterion of correctness. Namely, we shall accept a class C as a *model* of the specification S if the class C implements all the methods listed in the specification S and moreover it does it in such a way that all formulas of the set Ax are *invariants* of the class C.

The specification contains a few lines with the signature of operations (in Java it would be called an interface) and a few lines of invariants aka axioms. The invariants are algorithmic formulas.

We are showing that there exists a programmable model of the specification. By the more general property of algorithmic logic it follows that the specification S is free of contradictions i.e. it is consistent.

4.1. Informal description

The universe of a heap managing HM system consists of states and objects. A state may be viewed as a finite set of objects. For the purpose of the present work we can abstract from the structure of objects, of their types, even from their size. Therefore we shall speak of frames instead of objects. In order to make our presentation easier to follow, we abstract from the limitations on the size of states. This limitation is inessential one. [3)]

The universe of HM system consist of three sets

$$U = Frames \cup States \cup \{none\}$$

with the following operations:

$reserve : States \to Frames$
$insert : Frames \times States \to States$
$delete : Frames \times States \to States$
$member : Frames \times States \to \{true, false\}$
$initSt \in States$
$kill : Frame \times States \to States$

$States$ are finite sets of $frames$. For each state s function $reserve$ returns a frame f from outside the state s, that is f does not belong to s. For each pair $\langle e, s\rangle$ operation $insert$ returns the set-theoretical union of the set s, and the element e. Similarly, operation $delete$ returns the set s' that results by the deletion of element e from the set s. The element $initSt$ is the empty set.

4.2. Algorithmic theory ATHM of heap management

In this subsection we develop a specification of the Kreczmar's system in the form of an algorithmic theory, i.e. a theory based on algorithmic logic instead of first-order logic c.f. [Mirkowska and Salwicki 1987]. Our theory $\mathcal{ATHM}$ differs from the one proposed by Hanna Oktaba [Oktaba 1982] by the presence of functor kill and the corresponding axiom of kill.

[3]One can add these ingredients of object's size and its content later.

Each formalized algorithmic theory $\mathcal{T}$ can be identified with a triple $\mathcal{T} = \langle \mathcal{L}, \mathcal{C}, \mathcal{A} \rangle$, where $\mathcal{L}$ is a formalized algorithmic language, $\mathcal{C}$ is the syntactical consequence operation defined by the notion of proof. The last element of the triple is the set $\mathcal{A}$ of axioms specific for the theory $\mathcal{T}$. We can assume that the notion of *proof* is defined on the basis of the sets $\mathcal{A}_L$ of axioms and $\mathcal{R}$ the set of inference rules of algorithmic logic.
The formalized algorithmic language $\mathcal{L}$ of our theory $\mathcal{ATHM}$ consists of three sets of well formed expressions: terms T, formulas F, and programs P. The alphabet of the language contains the sets of variables, of functors, of logical functors, of program connectives, and auxiliary symbols like parentheses, commas, etc.
The set of algorithmic formulas is the least set of expressions that contains all first-order formulas over the same alfabet and is closed with respect to the usual formation rules.Moreover, for any program K and any algorithmic formula α, the expression $K\alpha$ is an algorithmic formula.

We shall consider variables of type F - for frames, usually denoted by $f, f', ...$ and of type S - for states,usually denoted by $s, s', ...$
The set of functors and predicates of the theory's language consists of:

$res : S \to F$
$amb : S \to F$
$ins : F \times S \to S$
$del : F \times S \to S$
$mb : F \times S \to \{true, false\}$
$kill : F \times S \to S$

and two constants $\mathtt{none} \notin \{F \cup S\}$ and $eS \in S$. The value of any variable f of type F is a frame, or $\mathtt{none}$.

The logical consequence operation $\vdash$ is defined as in [Mirkowska and Salwicki 1987]

Axioms specific of the theory $\mathcal{ATHM}$ are given below

HM$_1$) $\forall_{s \in S} \neg mb(res(s), s)$
For every state s, operation $res(s)$ returns a new frame, not an element of s

HM$_2$) $\forall_{f \in F} \neg mb(f, eS)$
the initstate eS is the empty set of frames

HM$_3$) $\forall_{s \in S} \left\{ \begin{array}{l} \textbf{while } s \neq eS \textbf{ do} \\ \quad s := delete(amb(s), s) \\ \textbf{od} \end{array} \right\} (s = eS)$
For every state s, the program while (above) terminates, hence, every state is a finite set of frames

HM$_4$) $\forall_{s \in S}\ s \neq eS \Rightarrow mb(amb(s), s)$
For every non-empty state, function amb returns a member of the state s

HM$_5$) $\forall_{f \in F} \forall_{s \in S} \{s' := ins(f, s)\}(mb(f, s') \wedge \forall_{f' \in F}(f' \neq f \implies mb(f', s) \Leftrightarrow mb(f', s'))$
operation ins adds frame f to the state s

HM$_6$) $\forall_{f \in F} \forall_{s \in S} \{s' := del(f, s)\}(\neg mb(f, s') \wedge \forall_{f' \in F}(f' \neq f \implies mb(f', s) \Leftrightarrow mb(f', s'))$
operation del deletes frame f from the state s

HM$_7$) $$mb(f,s) \Leftrightarrow \left\{ \begin{array}{l} \textbf{begin} \\ \quad s1 := s;\ bool := false; \\ \quad \textbf{while}\ s1 \neq eS \wedge \neg bool \\ \quad \textbf{do} \\ \quad\quad f1 := amb(s1); \\ \quad\quad \textbf{if}\ f = f1\ \textbf{then}\ bool := true\ \textbf{fi}; \\ \quad\quad s1 := del(f1, s1); \\ \quad \textbf{od} \\ \textbf{end} \end{array} \right\} bool$$

This formula defines the properties of relation member. It is **not** an implementation however. We postulate that the implemented cost should be constant.

HM$_8$) The operation kill is characterised by the axioms of the following scheme.
The index k may be any natural number $k > 0$, let $1 \leq i \leq k$.

$$\underbrace{((f_1 = ... = f_k) \wedge mb(f_1, s))}_{precondition} \Rightarrow \underbrace{[s' := kill(f_i, s)]}_{\text{statement}} \underbrace{(f_1 = ... = f_k = \textbf{none})}_{postcondition}$$

Any formula of this form is an axiom, it tells that operation kill in one move nullifies *all the references to the object pointed by the variable* f_i. And indeed, in the system of Kreczmar the cost of the operation kill is constant.

4.3. Applications of the specification

The above set of algorithmic formulas defines the requirements imposed on a class to be implemented. Moreover, it allows to prove some useful facts, i.e. the theorems of the ATHM theory.

THEOREM 4.1. *The program in the axiom HM$_7$ never loops, more precisely*

$$\{HM1-6\} \vdash \forall_{s \in S} \left\{ \begin{array}{l} \textbf{begin} \\ \quad s1 := s;\ bool := false; \\ \quad \textbf{while}\ s1 \neq eS \wedge \neg bool \\ \quad \textbf{do} \\ \quad\quad f1 := amb(s1); \\ \quad\quad \textbf{if}\ f = f1\ \textbf{then}\ bool := true\ \textbf{fi}; \\ \quad\quad s1 := del(f1, s1); \\ \quad \textbf{od} \\ \textbf{end} \end{array} \right\} true$$

For the proof see the Appendix B.

THEOREM 4.2. *For every state s, the following program terminates*

$$\textit{\{HM1-8\}} \vdash \forall_{s \in S} \left\{ \begin{array}{l} \textbf{begin} \\ \quad s' := eS; \\ \quad \textbf{while}\ s \neq eS\ \textbf{do} \\ \quad\quad r := res(s'); \\ \quad\quad \textbf{if}\ mb(r, s)\ \textbf{then}\ s := del(r, s)\ \textbf{fi}; \\ \quad\quad s' := ins(r, s') \\ \quad \textbf{od} \\ \textbf{end} \end{array} \right\} (s = eS)$$

i.e. the operation res: reserve a new frame, may replace the operation amb in the postulate that every state is a finite set of frames.

□

In our simplified world of references one may define the operation **new** as

f:= res(s); s:= ins(f,s);

i.e. reserve a new frame and include it into the set of reserved frames. We would like to attract the attention of the reader to the property of HM_1. One may use it to deduce the important fact

THEOREM 4.3. *Let T be any class, let (a_1, ..., a_k) be a list of actual parametres.*

$$\{\text{HM1-8}\} \vdash \textbf{\textit{new}}\ T(a_1, \ldots, a_k) \neq \textbf{\textit{new}}\ T(a_1, \ldots, a_k)$$

□

4.4. Properties of the specification

One can investigate the properties of the specification itself. We are able to state an important metatheorem about the system of axioms in HM. The following theorem was not formulated in [Cioni and Kreczmar 1984]. H. Oktaba proved a theorem on consistency for a similar set of axioms [Oktaba 1982], basically it was the set {HM_1 - HM_7 }.

METATHEOREM 1. *(on consistency of the set {HM1-8})*
The system of axioms HM_1 – HM_8 has a model.

For a sketch of the proof see the Appendix A. The model constructed in the proof will be called the *standard model*.

□

H. Oktaba proved another important fact:

METATHEOREM 2. *(representation theorem)*
Every two models of the axioms HM_1 – HM_7 are isomorphic, up to implementation of operations amb and res, to the standard model.

We agree that the operations res and amb may be implemented in several versions. It will suffice that their implementations satify requirements mentioned in the axioms HM_1 and HM_4.

However, to prove a similar metatheorem for the system HM_1 – HM_8 is a different and non-trivial task.

4.5. Variations of axiom's system

Are the simplifications we made important? One can easily observe two points:

— One can consider a slightly different operation reserve - with a parameter $appetite$ defining the size required for an object. This can be easily done by modification of the signature $res : S \times N \to F$ and leads to a new (consistent) set of axioms.
— Another extension of our system HM is defined when one describes the internal structure of an object. (The structure is determined by the declaration of class). This extension is also consistent.

Till now we needed not to introduce an operation of garbage collection. In our abstract version the set of Frames is isomorphic with the set of natural numbers.To make our theory more realistic we should introduce a postulate that the set of frames is finite.In this case a need arises of garbage collection.

One can ask how to express the property *the set Fr of frames is finite*? The answer is easy:

HM$_9$) $\exists_{s_0 \in St} \forall_{f \in Fr} mb(f, s_0)$
Which reads: *the set of frames is equal to some state, hence Fr is a finite set.*
The set of the axioms HM$_1$ – HM$_9$ is inconsistent. However, it is quite easy to repair it. We leave this as an exercise. **Hint.** Introduce a predicate $full$, a dual to the predicate $empty$.

5. FINAL REMARKS

Let us remark that since Java was introduced in 1995, the memory size has grown thousand times, from megabytes MB to gigabytes GB. The cost of garbage collection increased accordingly. For each algorithm of garbage collection must touch each cell of objects memory.

In the paper [Cioni and Kreczmar 1984] the memory management system is treated as a whole. The problems of garbage collection and dangling references were not separated. The safety question *does a given pointer points to an alive object* takes the central place of the system.

In Loglan'82 operation `kill()` is safely implemented with low, fixed cost. Each access to an object is checked, and it is done through three machine instructions only.

The heap management system offers five operations:

— creation of a new object – x := new T(),
— disposal of an object – kill(x),
— member(x) – verification if the value of a variable x is an alive object
— compact() – defragmentation of unused memory
— gc() – garbage collection

The frequency of garbage collection is reduced due to following discipline:

1° kill operation is called whenever an object should be deleted, the freed frame is added to the list of free memory frames,
2° during an operation of creation `new` object, the list of free memory frames is checked and used, prior to slicing a fragment of unoccupied memory (between the stack and the heap),
3° the operation of compactification (i.e. defragmentation) has priority over operation of garbage collection. (The cost of defragmentation is less than the cost of garbage collection.)

Open question.
Is it possible to construct the heap management system of better features (a cheaper or faster one)? An author of a (*verified*) answer will obtain a prize of 50 Euro.

ACKNOWLEDGMENT

The second author was financed by research fellowship within the Project *Information technologies: Research and its interdisciplinary applications*, Agreement number UDA POKL.04.01.01-00-051/10-00.

Appendix A - How to construct a model of ATHM theory?

In this section we gather remarks useful in the process of implementing the axioms/invariants of the theory ATHM.

METATHEOREM 1. (*on consistency of the set* {HM1-8}) The system of axioms HM1 – HM8 has a model.

This theorem does not appear in the paper [Cioni and Kreczmar 1984]. However, the construction of the heap management system and the proofs of invariants contained there, lead in a straightforward way to the proof of the metatheorem 1. We shall not repeat the detailed discussion, instead we present the main points. The presentation in [Cioni and Kreczmar 1984] is loaded with the details concerning the ways to treat the retrieved memory. For this presentation we shall assume that no limit is imposed on the memory and we

shall not explain how to organize the memory retrieved from deallocated objects. We hope that the reader is able to fill this gap.

Kreczmar observed that the question: *does a variable x points to a live object?* is the principal one. The notion of a dead object was known in the context of garbage collection. A dead object need not to be deleted immediately. It is enough to guarantee that it will be deleted in some future. And indeed in some programming languages the garbage collection may refuse to start immediately when called by *gc()* instruction, cf. [Aho et al. 2007], Kreczmar considered the whole life cycle of object.

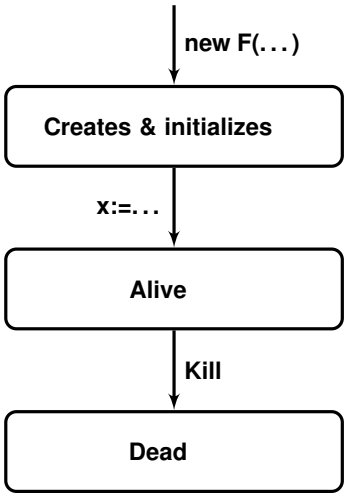

Fig. 1. Diagram of states of an object

Kreczmar remarked that the answer to the question: *is the object x alive?* is more important than the task of calculating the physical address of it.

Digression. Nobody is astonished that a compiler first checks whether the indexed variable A[i] exists and later calculates its physical address. In some languages this check is obligatory. **End of digression**

The first proposal is to use the concept of tombstones cf. [Gabbrielli and Martini 2010]. Remark that tombstones do not eliminate the error of memory leakage. Using tombstones one can retrieve the portions of memory previously occupied by objects. But how to retrieve the tombstones?

Kreczmar proposed another solution: object consists of a frame and a handle to it. The information contained in a handle allows to: 1°identify the frame (i.e. to calculate its physical address), and 2°answer whether the object is alive or not.

$$Objects = Frames \times Handles$$

Now, a handle h to an object o contains two pieces of information:

h[0] - a reference to the object, i.e. its physical address,
h[1] - an identification link of the object = serno.

Each object upon its creation obtains a unique, serial number. This number is stored in the object itself and also it is stored in the handle, as its second element $h[1]$.

In the table II below we are sketching the implementation of the operations creation, disposal, member. Operation create is defined as the composition of operations reserve and insert. Similarly operation disposal uses the operation delete.

Now, the proof of METATHEOREM 1 may proceed by the induction w.r.t. the number of executed operations *create, access, disposal*. The thesis we are going to prove is *Let the value of a variable* x *is* $\langle b, key\rangle$. *Then* $x \neq none$ ***iff*** *key = lock i.e. =Memory[b+1]*

Table II. High points of Implementation

action	**example**	**code**
creation	x:=new T(length)	*find a piece of free memory of the size length, and second of size 2. Let* fr *and* h *be their addresses respectively*. new object is <fr, h> ; fr[0] $\leftarrow$ $length$; h[1]$\leftarrow$ $serno$; h[0] $\leftarrow$ fr; $x[0] \leftarrow h$; $x[1] \leftarrow serno$; $serno \leftarrow serno$ +1
access (is alive?)	$x.attr$	$h \leftarrow x[0]$; **if** $h[1] = x[1]$ **then** $addr \leftarrow h[0]$ **else** signal exception*ReferenceToNone* **fi**; **return** $addr[attr]$
disposal	kill(x)	$h \leftarrow x[0]$; $h[1] \leftarrow h[1] + 1$ *and manage the retrieved memory*

Appendix B - proof of Theorem 4.1

The aim of this section is to show the advantage of specifications over interfaces. Profits of specification follow from the possibility to use it as a base for formal proof of an application. The proof uses the calculus of algorithmic logic.

THEOREM 4.1 The program in the axiom HM_7 never loops, more precisely

$$\{HM1-6\} \vdash \forall_{s\in S} \left\{ \begin{array}{l} \textbf{begin} \\ \quad s1 := s;\ bool := false; \\ \quad \textbf{while}\ s1 \neq eS \wedge \neg bool \\ \quad \textbf{do} \\ \quad\quad f1 := amb(s1); \\ \quad\quad \textbf{if}\ f = f1\ \textbf{then}\ bool := true\ \textbf{fi}; \\ \quad\quad s1 := del(f1, s1); \\ \quad \textbf{od} \\ \textbf{end} \end{array} \right\} true$$

PROOF. The proof takes only 10 steps. We start with the axiom HM3. In each step we are using one axiom and/or one rule of algorithmic logic. All these tools are quoted below for the convenience.

Ax_{19}	$\textbf{begin}\ K; M\ \textbf{end}\,\alpha \equiv K\,(M\,\alpha)$.
Ax_{18}	$((x := \tau)\gamma \equiv (\gamma(x/\tau)$
R_1	$\dfrac{\alpha, (\alpha \Rightarrow \beta)}{\beta}$
R_2	$\dfrac{(\alpha \Rightarrow \beta)}{(K\alpha \Rightarrow K\beta)}$
aux_1	$\dfrac{(\alpha \Rightarrow \beta)}{\textbf{while}\ \beta\ \textbf{do}\ K\ \textbf{od true} \Rightarrow \textbf{while}\ \alpha\ \textbf{do}\ K\ \textbf{od true}}$
aux_2	$\dfrac{\textbf{while}\ \alpha\ \textbf{do}\ K\ \textbf{od true}, (M; K\,\alpha) \equiv K\,\alpha}{\textbf{while}\ \alpha\ \textbf{do}\ K; M\ \textbf{od true}}$
aux_3	$\dfrac{\alpha,\ M\,true}{M\,\alpha}$

The formal proof of the theorem is given below.

1) $\forall_{s1\in S}\left\{\begin{array}{l}\textbf{while } s1 \neq eS \textbf{ do}\\ \quad s1 := delete(amb(s1), s1)\\ \textbf{od}\end{array}\right\}(s1 = eS)$ *axiom* HM_3

let us denote the program by K

2) $(s1 = eS) \Rightarrow true$ *tautology*

3) $\forall_{s1\in S} K(s1 = eS) \Rightarrow \forall_{s1\in S} K\, true$ $from\ 2)\ by\ R2\ rule$

4) $\forall_{s1\in S}\left\{\begin{array}{l}\textbf{while } s1 \neq eS \textbf{ do}\\ \quad s1 := delete(amb(s1), s1)\\ \textbf{od}\end{array}\right\} true$ $from\ 1)\ and\ 3)\ by\ R1$

5) $\{s1 := delete(amb(s1), s1)\}\,\alpha \equiv \{f1 := amb(s1); s1 := delete(f1, s1)\}\,\alpha$ *by* $Ax18$

α is any formula that does not contain the variable $f1$

6) $\forall_{s1\in S}\left\{\begin{array}{l}\textbf{while } s1 \neq eS \textbf{ do}\\ \quad f1 := amb(s1);\ \ s1 := delete(f1, s1)\\ \textbf{od}\end{array}\right\} true$ $from\ 5)\ and\ 4)$

7) $\forall_{s1\in S}\left\{\begin{array}{l}\textbf{while } s1 \neq eS \textbf{ do}\\ \quad f1 := amb(s1);\ \ s1 := delete(f1, s1)\\ \quad \textbf{if } f = f1 \textbf{ then } bool := true \textbf{ fi};\\ \textbf{od}\end{array}\right\} true$ *by* $aux1$ *rule*

8) $\forall_{s1\in S}\left\{\begin{array}{l}\textbf{while } (s1 \neq eS) \wedge \neg bool \textbf{ do}\\ \quad f1 := amb(s1);\ \ s1 := delete(f1, s1)\\ \quad \textbf{if } f = f1 \textbf{ then } bool := true \textbf{ fi};\\ \textbf{od}\end{array}\right\} true$ *by* $aux2$ *rule*

9) $\forall_{s\in S}\left\{\begin{array}{l}s1 := s;\ bool := false;\\ \textbf{while } (s1 \neq eS) \wedge \neg bool \textbf{ do}\\ \quad f1 := amb(s1);\ \ s1 := delete(f1, s1)\\ \quad \textbf{if } f = f1 \textbf{ then } bool := true \textbf{ fi};\\ \textbf{od}\end{array}\right\} true$ *by* aux_3 *applied twice*

10) $\forall_{s\in S}\left\{\begin{array}{l}\textbf{begin}\\ \quad s1 := s;\ bool := false;\\ \quad \textbf{while } (s1 \neq eS) \wedge \neg bool \textbf{ do}\\ \quad\quad f1 := amb(s1);\ \ s1 := delete(f1, s1)\\ \quad\quad \textbf{if } f = f1 \textbf{ then } bool := true \textbf{ fi};\\ \quad \textbf{od}\\ \textbf{end}\end{array}\right\} true$ *by* $Ax19$

□

REFERENCES

A. Aho, M. Lam, R. Sethi, and J. Ullman. 2007. *Compilers: principles, techniques & tools*. Addison Wesley, Boston. 1009+26 pages.

A. W. Appel. 1998. *Modern Compiler Implementation in Java*. Cambridge University Press. http://www.bibsonomy.org/bibtex/2da0c2a0f1085a376345e0379ef0886e1/dblp

J. Barnes. 1996. *Programming in Ada 95*. Addison-Wesley Publ., Boston. 720 pages.

L. Cardelli, J. Donahue, L. Glassman, J. Mick, B. Kalsow, and G. Nelson. 1989. *Modula-3 report*. Technical Report 52. Xerox.

G. Cioni and A. Kreczmar. 1984. *Programmed Deallocation without Dangling References*. Information Processing Letters 18 (1984), 179–185.
http://lem12.uksw.edu.pl/wiki/Programmed_dealocation_without_dangling_reference. (1984). [Online; accessed 29-October-2014].

O.-J. Dahl. 1974. *On garbage collection*. personal communication to A.S.. (June 1974).

C.N. Fisher and R.J. Leblanc. 1980. *The implementation of run-time diagnostics in Pascal*. IEEE Trans. Softw. Eng. 6 (1980), 313–319.

M. Gabbrielli and S. Martini. 2010. *Programming Languages: Principles and paradigms*. Springer.

J. Gosling and H. McGilton. 1995. *The Java Language Environment*. Technical Report. 30 pages.

J. Damato and A. Gupta 2010. *Garbage Collection and the Ruby Heap Presentation*. Presented as RAILSCONF 2010, Baltimore, Maryland.

K. Jensen and N. Wirth. 1974. *Pascal user manual and report*. Springer Verlag, New York.

A. Kreczmar. 1987. *A short introduction to the new running system written in Loglan'82*.
http://lem12.uksw.edu.pl/Loglan82/Doc/rsloglan-in-Loglan.pdf. (1987). [Online; accessed 1-October-2014].

D.B. Lomet. 1975. *Scheme for invalidating references tofreed storage*. IBM Journ. R & D 19 (1975), 26–35.

G. Mirkowska and A. Salwicki. 1987. *Algoritmic Logic*. PWN & Reidel, Warszawa & Dordrecht. 367 pages.
http://lem12.uksw.edu.pl/images/3/35/Algorithmic_Logic.pdf. [Online; accessed 28-November-2014].

H. Oktaba. 1982. *On algorithmic theory of references*. Ph.D. Dissertation. Institute of Informatics, University of Warsaw.
[see also Mirkowska and Salwicki 1987, pp. 328 - 341].

A. Salwicki. 2013. *Loglan'82*.
http://lem12.uksw.edu.pl/wiki/Loglan'82project. (2013). [Online; accessed 2-November-2013].

B Stroustrup. 2013. *The C++ Programming Language*. Addison-Wesley Publ., Boston. 387 pages.

Wikipedia. 2013a. Python programming language — Wikipedia, The Free Encyclopedia. (2013). http://en.wikipedia.org/w/index.php?title=Python(programminglanguage)&oldid=552039830 [Online; accessed 29-October-2013].

Wikipedia. 2013b. Reference counting — Wikipedia, The Free Encyclopedia. (2013). http://en.wikipedia.org/w/index.php?title=Reference_counting&oldid=552039830 [Online; accessed 11-May-2013].

Wikipedia. 2013c. Weak reference — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Weak\reference\&oldid=552039830. (2013). [Online; accessed 11-May-2013].