Safe Low-Overhead Memory Management for Concurrency and Parallelism

Denys Shabalin, I&C, EPFL

Abstract—This report surveys three papers related to safe memory management. The first two papers expose two different approaches that rely on static compile-time annotations to enhance performance of memory management at runtime. The third paper shows how to implement an efficient garbage collector that has impressively low pause times and never stops all the threads at the same time.

Index Terms—thesis proposal, candidacy exam write-up, EDIC, EPFL

I. INTRODUCTION

Memory management strategy is a critical part of any modern programming language. The choice of a particular model not only influences performance characteristics but also changes the way people reason about the programs in a given language.

Most existing high-level languages prefer to completely eliminate the memory management burden from the end user by providing a fully garbage collected environment. In such languages, developers don’t have means to convey their memory management preferences and must rely solely on the garbage collector to perform the memory management decisions. Depending on the implementation strategy, language implementors often has to make sacrifices in terms of either latency or throughput performance.

Such limitations have caused people to develop alternative memory management solutions to sidestep GC limitations that are commonly known as off-heap memory. Objects allocated off heap are invisible to the GC and can be managed directly by the programmer.

On the other side of the memory management spectrum, there is a small group of languages like Cyclone [1] and Rust [11] that have been developed with an explicit goal of safe low-overhead memory management. These languages sacrifice end user experience by prioritizing runtime efficiency over notational convenience.

In this paper, we overview three memory management paradigms:

- Region-based memory management
- Ownership-based memory management through uniqueness types
- Reference-counted garbage collection with automatic cycle detection

The paper is organized as follows. We survey each of the three aforementioned memory management models outlining their strengths and weaknesses in sections II, III and IV correspondingly. In section V, we propose a plan on how the best features of these models can be combined to advance the state of the art of memory management.

II. REGION-BASED MEMORY MANAGEMENT IN CYCLONE

Cyclone is a systems programming language based on C that provides statically-checked safe memory management. It introduces a number of additional features such as parametric polymorphism, subtyping, tagged unions and most importantly regions.

Regions in Cyclone are based on the work of Tofte and Talpin [3] who originally explored the concept in context of the ML family of functional programming languages.

Regions denote explicitly delimited scopes that let one bind memory allocations to them. Once the execution leaves the scope, all of the allocations performed in the scope are automatically cleaned up.

Any reference that points to region-allocated memory is valid only during the period of time when the region is open and becomes a dangling pointer if that reference ever manages to escape the region scope. Extra care has to be taken to ensure that such references are never dereferenced. Cyclone manages to statically ensure this property thanks to its type system.
A. Regions

Regions in Cyclone come in three distinct flavors:

- **The heap region.** Heap corresponds to a forever-alive unique region that stores global static data. Allocations performed there exist forever and are never reclaimed by default (e.g. one can not call free on such allocations.) Cyclone also optionally supports the use of BoehmDemers-Weiser garbage collector [2] to clean up the heap.

- **Stack regions.** These regions unify stack allocation with dynamic allocation by representing it as just another type of region. The lifetime of the stack regions is defined by the lifetime of the corresponding lexically enclosing blocks.

- **Dynamic regions.** Dynamic regions are explicitly-delimited dynamically-sized areas of memory that allow one to perform unbounded number of allocations.

Despite the differences in storage models of the different regions, references to region-allocated objects do not require any additional runtime information except a raw machine pointer to the corresponding memory location.

This is possible due to the fact that region-related metadata exists only at compile time and is completely erased at runtime. For example:

```c
struct Point *r;
printf("Point at (%d, %d) ",
p->x, p->y);
```

Here `p` is a point allocated in region `r`. The type of `p` contains additional static information about the region where it was allocated. This information can often be inferred automatically.

To propagate this information safely along the program, Cyclone supports region polymorphism that lets one abstract over any concrete region. If we were to refactor message-printing into a separate subroutine it would look like:

```c
void printPoint<r>(struct Point*r p) {
    printf("Point at (%d, %d) ",
p->x, p->y);
}
```

If the user doesn’t explicitly specify region parameters for function argument types, Cyclone introduces additional ones automatically. This default helps reduce a number of explicit region polymorphism annotations, especially when porting existing C code.

Although region membership information is available in all pointer types it’s not enough to ensure safety from dangling pointers in presence of language features that let one hide information such as existential types and universal polymorphism.

B. Effects

To ensure safety in presence of features that let references escape their defining scope, Cyclone uses a simple effect system to statically guarantee that no dangling pointer is ever going to be dereferenced.

The effect system contains a single type of effect that represents a capability to access region with given name. This capability is required to perform any reads and writes on a pointer.

For example, if we were to define a function that computes the distance between two points:

```c
float distance<r1, r2>(struct Point*r1 p1, struct Point*r2 p2);
```

It would require a capability to read both `r1` and `r2` that is denoted by `{r1, r2}` effect. Effects can be specified manually to explicitly state the regions that are expected to be touched by the function. By default, effect of a function is automatically inferred conservative bound on the regions function might access.

Although Cyclone has effects, it doesn’t provide a way to introduce effect variables or have any other means to abstract over effects. Instead, a simple built-in operator `regions_of(t)` is used to extract all regions of a given type. This operator in combination with region polymorphism lets one to represent complex types which would have required effect variables otherwise.

C. Type system

The paper includes an excerpt of the formal system that was designed to prove soundness of the Cyclone’s region system. The formalisation is based on the following core judgements:

- **Expression typing:** $\Delta, \Gamma; \gamma; \epsilon \vdash e : \tau$
- **Region liveness:** $\gamma \vdash \epsilon \Rightarrow \rho$
- **Statement typing:** $\Delta, \Gamma; \gamma; \epsilon; \tau \vdash stmt s$
- **Statement termination:** $\gamma; \epsilon \vdash \tau \Rightarrow \epsilon$

Where $\Delta$ contains type and region variables in the current scope, $\Gamma$ corresponds to value environment, $\gamma$ stores partial order on region lifetimes and lastly $\epsilon$ corresponds to effect environment. $\tau$ in expression typing returns the type of the given expression in given context, $\tau$ in statement typing checks that return expression has correct type.

Let’s have a look at select few rules that illustrate the mechanics of the type system.

$$
\Delta, \Gamma; \gamma; \epsilon \vdash e : \tau \Rightarrow \rho \quad \gamma \vdash \epsilon \Rightarrow \rho \\
\Delta, \Gamma; \gamma; \epsilon \vdash e : \tau
$$

This rule specifies type checking of the pointer dereference. To dereference a pointer, an expression must have a pointer type and its region must be alive according to current effect environment.

Another notable rule represents typing of the function applications:

$$
\Delta; \Gamma; \gamma; \epsilon \vdash e_1 : \tau_1 \Rightarrow \epsilon_1 \quad \Delta; \Gamma; \gamma; \epsilon \vdash e_2 : \tau_2 \Rightarrow \epsilon \Rightarrow \epsilon_1 \\
\Delta; \Gamma; \gamma; \epsilon \vdash e_1(e_2) : \tau
$$
To call a function, an expression on the left must return a function type with effects $\epsilon_1$, argument type must match function argument type $\tau_2$ and, lastly, effects in current environment $\epsilon$ must subsume effects that are required for given function $\epsilon_1$.

D. Conclusion

Cyclone’s experience proves that region-based memory not only works for functional programming languages like ML but can also be adapted to fit safe low-level systems programming with direct control over memory layout. Cyclone succeeds at providing strong static guarantees and makes it easy to port existing C applications by providing familiar syntactic and semantic model that should be easy to grasp for experienced systems programmers.

Nevertheless, Cyclone is still quite low-level language and might not be a perfect fit for developers who are used to higher-level languages like Java or C# that provide a convenience of a fully garbage-collected environment. Syntactic overhead of region annotations is low, but it’s still there and requires a conscious effort to get annotations right.

III. UNIQUENESS AND REFERENCE IMMUTABILITY FOR SAFE PARALLELISM

The C# programming language has been a prolific substrate for research, and one of the latest developments is [4] that introduces type qualifiers to control mutability and aliasing to ensure safety of parallel programs.

A. Reference qualifiers

The main contribution of the paper is introduction of reference type qualifiers and the accompanying type system to ensure safety of the properties they represent.

In this system references can be:

- **writable**. Corresponds to regular unconstrained object reference that doesn’t impose any limitations on reads and writes.
- **readable**. A reference that constraints writes through current reference. Other aliasing writable references might exist to the same memory. All object graph traversals of such references produce either readable or immutable results.
- **immutable**. A reference to immutable object cluster. All objects within the cluster must be immutable and no aliasing non-immutable references may exist to it.
- **isolated**. A reference to unique object cluster. All of the references within the cluster are isolated to the references within the cluster. No aliasing references to any of the parts of the cluster might exist with an exception of references to immutable objects.

Isolated and immutable qualifiers are cornerstone tools that let one safely share data in parallel environment without a risk of interference between paralleling running threads that could have caused non-determinism and hard to reproduce bugs that depend on scheduling of threads in particular environment.

B. Isolated conversion and recovery

Unlike other systems that require special support to accommodate borrowing semantics of unique references, this work presents novel approach that lets one to temporarily surrender uniqueness information and downcast the reference to more permissive readable and writable qualifiers.

As long as the operations performed on the isolated references are local to the scope of the code the system is able to automatically recover back to isolated qualifier without any extra notational overhead. This permits safe local mutation of mutable isolated object clusters without surrendering of isolated property of the cluster.

```c
isolated IntBox increment(isolated IntBox b) {
    // implicitly convert b to writable
    b.value++;
    // convert b *back* to isolated
    return b;
}
```

Isolated references can also be easily converted to immutable ones. Due to uniqueness of the reference it’s possible to convert all of the isolated graph into immutable one at once.

In situations when isolated references escape to context where uniqueness may not be recovered the system is able to propagate that information back in a flow-sensitive manner ensuring that it’s impossible to get multiple inconsistent references to the same memory. For example:

```c
isolated IntList l = ...;
// implicitly updates l’s permission
// to immutable
immutable IntList l2 = l;
// Type Error!
l.head = ...;
```

Here creation of two distinct incompatible references (immutable and writable) to the same data is caught by flow-sensitive typechecking.

C. Type system

A formal version of the system was implemented to model flow-sensitive typechecking and prove its soundness. The formal system provides a simple core calculus with minimal feature set. Advanced features are discussed separately to clearly illustrate core ideas in isolation.

The system introduces a number of abbreviations for core concepts of the languages. Some of the more important ones are listed below:

- Commands $C$. Statement or combination of statements.
- Class types $T$ with corresponding class definitions $TD$ that can contain fields $fld$ and methods $meth$.
- Permissions $p$. One of the four reference type qualifiers we’ve discussed before.
- Types $t$. Can either be primitive ($int$ and $bool$) or permission-qualified class types $p.T$.
- Environment $I$ that binds names to types in current scope.

The type system is build as a composition of the following core judgments:
Rust [11] uses ownership types with explicit borrowing as the main foundation for memory management. For example in concurrent GC and other runtime-related optimisations is the previous reference gets to be overwritten reference to the other. Because only one reference may exist at any given time the previous reference becomes isolated. The collector is run on the objects that are suspected to be part of dead cycles that are the roots of the cycle detector. The synchronous collector on the other side implements an efficient cycle collection in reference counted systems [8] is one of the first papers on the Recycler garbage collector. It introduces two algorithms for cycle collection in reference counted garbage collection setting: one synchronous and one concurrent. As we’ll see later the concurrent version didn’t quite stand up to the test of time and has been later replaced with the concept of sliding views.

Efficient on-the-fly cycle collection [5] presents the latest iteration of research on reference counted garbage collectors that was done under IBM’s Recycler project. To explain the main contribution of the paper we need to briefly overview work on collectors that preceded it as many of the concepts are re-used from previous publications.

### A. Previous work

**Concurrent cycle collection in reference counted systems**

The main intuition for applying uniqueness types to memory management relies on the fact that as soon as one looses the handle to unique object cluster, the memory can be safely deallocated as no other references to the cluster may exist.

This work slightly complicates the intuition due to the fact that isolated references can contain references to immutable object clusters. This feature interaction conceptually requires garbage collection. It’s possible to either disallow such references or provide optional garbage collector to support this use case.

### E. Conclusion

This paper presents a fresh look at ownership and immutability types in context of safe parallelism in object-oriented languages. The system provides comprehensible framework with support for advanced features like polymorphism over both types and qualifiers with annotations that are fairly easy to comprehend.

On the other hand, similarly to Cyclone, annotations introduce non-trivial notational overhead that might be hard to grasp for beginner developers. As soon as standard library requires such annotations it means that all of the code that uses it gets transitively infected and one must be conscious of the reference qualifiers at all times.

### IV. AN EFFICIENT ON-THE-FLY CYCLE COLLECTION

The flow-sensitive nature of the type system is expressed through the core typing relation that can change the environment as typechecking proceeds through commands. This allows one to express changes of permissions on variables in flow-dependent manner.

Select few core typing rules are given in Figure 1. Unlike comparatively simple rules we’ve seen in Cyclone’s type system before, here one needs to specially handle reads and writes depending on the permission of the reference types that are currently being worked with.

Consuming read (T-FieldConsume) is the good example of how one performs ownership transfer from one isolated reference to the other. Because only one reference may exist at any given time the previous reference gets to be overwritten with null.

The paper doesn’t go into details on applications of type qualifiers for memory management. C# relies on the .NET GC to automatically clean up memory. A possibility of using immutable annotations to reduce the cost of write barriers in concurrent GC and other runtime-related optimisations is briefly discussed in the paper.

Other systems have successfully used uniqueness types as the main foundation for memory management. For example Rust [11] uses ownership types with explicit borrowing in an environment without garbage collection.

### D. Memory management

The collector is run on the objects that are suspected to be a part of dead cycles that are the roots of the cycle detector. Object is suspected if its reference count has been decremented. Such objects are put into Roots buffer that is used as a starting point for the cycle detector. Transitive closure of Roots is treated as a single graph and traversed once to find cycles. The traversal colors nodes in the graph to discover if they can be part of cycle. Whenever a definitive cycle is found it’s reclaimed immediately with all its children.

Extra care is taken to not spend time on walking objects graphs that are inherently acyclic (such as scalars, references.
to final acyclic classes and arrays of the two.) Treating such objects specially decreases number of cycle candidates and time of the Roots graph traversal.

An on-the-fly reference counting garbage collector for Java [6]. This paper introduces an important concept of the sliding view that lets one perform cycle detection in the reference counted environment without stopping all the threads at the same time. Such collectors are also known as on-the-fly collectors.

Unlike most other reference counted systems this paper doesn’t maintain up-to-date reference count at all times but rather only at collection times. This effectively coalesces the work needed to maintain the count. Another noteworthy detail is the fact that reference counts are only maintained for heap-to-heap references. Stack/register-to-heap references are not counted.

To introduce the sliding view algorithm, first a simpler stop-the-world version of it called snapshot algorithm is introduced. To maintain reference counts between collections the algorithm maintains the old values of all written fields before it was first modified between collections. This recording is implemented through a novel synchronization-free write barrier. The collector start its traversal based on values of recorded modified fields to find cyclic garbage and update reference counts along the traversed part of the object graph. Whenever reference count of an object reaches zero it’s not immediately reclaimed but rather put on ZCT (zero-count table) to be considered for reclamation later. Because stack/register-to-heap reference counts are not maintained, collector needs to ensure that object is not referenced by the roots before reclaiming it.

The sliding view algorithm builds upon snapshot algorithm by making it only stop one particular thread a time. This complicates the implementation as the view of the object graph is not precise but rather a fuzzy non-atomic sliding view over the length of the garbage collection cycle. To ensure safety in such conditions an additional snoop mechanism is introduced. It remembers all objects to which a new reference is introduced during collection interval of time and effectively makes them additional dynamic set of roots not to be collected at current cycle.

Age-oriented concurrent garbage collection [7]. Age-oriented garbage collection is an adaptation of the idea of generational collector for on-the-fly setting. Because on-the-fly collectors maintain quite good pause time characteristics on their own the goal of the work is to ultimately improve throughput of the collection.

The age-oriented collector is defined as collector that:

- Always collects the entire heap
- During a collection treats each generation differently

The purpose of segregation of objects into different generations is to allow one to apply algorithms that are more efficient for particular generation. This paper chose to use mark-and-sweep for young generation and on-the-fly reference-counted garbage collector for the old generation.

Collecting entire heap has a benefit of not having a hard limit on the size of the young generation. Because pause times are not dependent on the size of the workload in the on-the-fly setting this improves throughput by not having to perform collections of young generation very often. Additionally, this simplifies treatment of pointers that traverse generation boundaries as generations can be determined dynamically during collection.

B. Contribution of the paper

Efficient on-the-fly cycle collection builds upon the cycle collector of [8]. It doesn’t use the concurrent version of the collector but rather adapts the synchronous version to fit into sliding view [6] on-the-fly collection setting. Asynchronous version of the collector in [8] is criticized for adding too much overhead in order to ensure safety in concurrent setting.

Combination of the two approaches is not trivial as the sliding views framework elides most reference count updates that happen between collection cycles that the collector from [8] can not handle by default. This necessitates an extension of the sliding views framework with extra analysis to allow cycle collector to base its candidate on the decrements that are being recorded combined with special treatment for freshly allocated objects.

Additionally the work adapts the age-oriented approach of [7] to reduce cost of reference counting in old generation by using a tracing mark-and-sweep collector on the young generation. Experimental results demonstrate that this indeed decreases amount of work cycle collector needs to handle.

C. Conclusions

The paper presents a latest iteration of research on on-the-fly reference-counted garbage collection. It combines best of the ideas from the collectors that preceded, most interestingly the sliding views [6] approach to reference counting. This allows them to achieve impressive sub-2ms pause times that never stop more than one thread at any given point in time.

Despite good latency characteristics the collector still struggles with young objects and requires additional tracing collector for early generation to handle influx of objects with high mortality rate. They conclude that on modern hardware reference counting is only suitable for large old generations with low mortality rates.

V. RESEARCH PROPOSAL

At the moment there is a clean split between low-level systems programming memory management strategies and high-level garbage-collected ones. The first sacrifice language’s ease of use by introducing additional features to deal with memory management. The second optimize ease of use at the cost of additional memory management overhead.

The cost of garbage collection in high-level languages often drives people to use alternative off-heap memory models that are usually poorly supported by the language and provide much weaker safety guarantees (e.g. sun.misc.Unsafe on JVM). This is caused by the fact that there is no way to
communicate programmer’s intent to the underlying garbage collector. Garbage collection has to be implemented assuming the worst-case scenario through expensive techniques such as write barriers that sacrifice performance.

We believe that next generation memory management systems need to both provide reasonable hassle-free defaults for beginners and expose additional tools for expert developers to obtain control over memory management for performance-sensitive pieces of code.

Initial work in that direction has been done as a part of scala-offheap project [12]. It builds on top of language extension primitives such as macros [9] and values classes [10] to expose Cyclone-style region-based memory-management to Scala. We were able to closely model the static checking model without changing the underlying language using implicit values as means to encode necessary effect checking. Unfortunately due to some unpleasant feature interactions and heavy notational overhead we decided not to provide those static guarantees in current publicly released version of the library.

In the future, we plan to experiment with integrating user-guided memory management schemes like region-based memory into garbage-collected environment. Our current implementation runs on the unmodified JVM and thus can’t support pointers between region-managed heap and garbage collected heap as garbage collector is not aware of its existence. Use of a variation of region-based memory in combination with a garbage collector will effectively provide programmable early generations that programmer can control to optimize applications for their domain. On-the-fly reference counting schemes, like the one given in [5], are a possible option for old generation collection that only needs to handle long-living objects.

REFERENCES