Advances in Programming Languages: Regions

Allan Clark and Stephen Gilmore
The University of Edinburgh

February 22, 2007

Introduction

The design decision that memory will be managed on a per-language basis by a garbage collector is an advance in programming languages. Programmer errors of several different kinds are prevented when memory is not manipulated via the malloc/free model. The fact that mainstream languages such as Java and C# depend on a garbage collector indicates that this memory management model has found widespread acceptance, and is considered reliable enough and efficient enough for many applications.

Predictability of memory use

For all of its benefits, an identifiable problem with garbage collection is that it becomes difficult for developers to predict the memory use of their programs, because de-allocation points are not explicit in the program. Where allocation and de-allocation is explicit, as with `malloc` and `free`, developers often have an understanding of how control is (meant) to flow from the point of allocation to the point of de-allocation.

Allied to the problem of predicting memory consumption of a garbage-collected program is the problem of estimating the time spent garbage collecting. There are two parts to this. Firstly, how often, and *when* will the collector run? Secondly, when it runs how long will it take, the *pause time*. For some application areas, such as *real-time*, this uncertainty is enough to prevent garbage-collected languages from being used.

Tagging

A further unfortunate side effect of garbage collection is that the data is often required to be *tagged* with information describing its type. The garbage collector must be able to tell when a value is a pointer or simply a word size value. Otherwise the collector may try to follow an invalid pointer. Furthermore the collector must know the size of value pointed to, so that it may follow the pointers contained within that heap value. This means that the programmer has less control over the representation of data.

Under the *pure stack discipline* of earlier languages such as Algol 60 memory use was even more predictable because memory was only stack allocated (the run-time had no heap). Data structures were allocated on entry to a function and de-allocated at the end. The disadvantage of this discipline though is that the size of the data structure to be allocated needs to be known on entry to the function which allocates it, which is sometimes inconvenient.

What would be the perfect memory management scheme? We would like a malloc/free-like discipline, where the free statements are automatically inferred for us by the compiler. We would like a safety guarantee that no dangling pointers are dereferenced. However we would also like to be able to assure ourselves that a particular object is freed no later than a particular point.

Regions

The *region* concept in memory management is an attempt to combine the safety of garbage collection with the predictability of stack allocation. Regions are an experimental idea in programming language design: no mainstream language in widespread use presently provides them. Even minority programming languages outside the mainstream do not all provide regions: O'Caml does not provide regions. Cyclone does, so all of our program examples which use regions will be in the Cyclone language.

The idea of region memory management is that a stack of regions is maintained. Creation of a new region places that new region on the top of the stack. A region may be de-allocated only when it is at the top of the stack. Because of this de-allocation is generally syntax driven, for example with one region per function or block. In this sense region memory management is similar to a stack allocation discipline.

However, memory for new data structures, may be allocated within *any* region. So in particular a function may return an object of arbitrary size, simply by allocating it in a region which outlives the stack frame of the function. The compiler can check that we do not dereference a pointer stored within a region that is not live at the time of the dereference.

Lexical regions

The most common form of regions are *lexical regions*: one created within a block of code. All the regions created in a block are de-allocated once the end of that block of code is reached. Because blocks are nested, the LIFO restriction on the lifetime of regions is maintained automatically.

Blocks also have a stack-allocated region associated with them, in which local variables can be stored. These regions can be implemented faster, but retain the restrictions of pure stack allocation, in that the regions are not dynamically sized.

```
int *‘r bar(region_t<‘r> r, int * x)
{
  int y = *x + 42;
  return rnew(r) y;
}

void foo () {
 B:{ region <‘r> h;
     int *x = rnew(h) 42;
     int *y = bar(h,x);
   }
}
```

In this example, we create a region `h`, and store a value `x` into that region. We also pass the region into the function `bar`, which may then allocate some values—in particular the value pointed to by the return pointer—into the region `h` before returning. All of the values allocated into the region `h`—including those allocated by the function `bar`—are de-allocated when the block `B` goes out of scope.

Safety

Cyclone rejects at compile-time ill-typed functions such as the following:

```
int *‘L bad() {
 L: { int x = 3;
      return &x;
    }
}
```

The value x is declared inside the block L, and hence it is stored in the region associated with that block. The address of x is therefore an invalid pointer anywhere outside block L.

Through its use of regions Cyclone prevents the *stale address* problem which we encountered with C and C#.

```
/* File: programs/cyclone/StaleAddresses.cyc */
#include "stdio.h"
#include "stdlib.h"

int* makeCounter() {
  int counter = 0;
  return &counter; /* return address of local variable */
}
int main() {
  int* p;
  p = makeCounter();
  printf("*p is %d\n", *p); /* prints "*p is 0" */
  exit(0);
}
```

This is rejected at compile time with the following message:

```
StaleAddresses.cyc:7:
   returns value of type int *‘makeCounter
            but requires int *
   ‘makeCounter and ‘H are not compatible.
```

This is a further example of an ill-typed function rejected by Cyclone at compile time:

```
int *‘r bar(region_t<‘r> r, int * x)
{ return rnew(r) (*x + 42); }

int foo () {
  int *y;
  B:{ region <‘r> h;
      int *x = rnew(h) 3;
      y = bar(h,x); /*Error here*/
    }
  return *y;
}
```

The error in this program is that it tries to make the region-allocated pointer x outlive its region by assigning it to a region with a longer lifetime.

Region subtyping

The LIFO nature of region life times, suggests a subtyping relation on region types. A region R1 is a subtype of R2, if R1 outlives R2. This can be useful, for example when we want to choose between two values, as in this example:

```
void decide (int b, int *‘p1 p1, int *‘p2 p2)
{ L : { int *‘L p;
        if (b) p = p1; else p = p2;
        *p = 3;
        /* do something more interesting with p */
       }
}
```

Advantages of using regions

There is no unpredictable runtime cost for region memory management, caused by a dynamic garbage collector. Region management has a low overhead. The programmer can control how long allocated values will live. Whole regions can be de-allocated at once. This can often result in a speed-up over manual memory management. Many region annotations can be automatically inferred by the compiler, hence the burden of manual memory management to the programmer can be largely lifted.

Additions

We add extra *kinds* of regions, these regions are not held on the region stack, and are managed differently.

- The first is a garbage collected heap.
- Unique regions, where each value in a unique region is pointed to by a *single* pointer.
- Dynamic regions, which can be dynamically de-allocated at (almost) any program point.
- Finally, a reference counted region.

Unique pointers

The unique region holds values which can be de-allocated at any time. To avoid dangling references, a static analysis ensures that values in the unique region can only ever be accessed through one pointer. In other words, there are no aliases of that pointer. When an object is freed, the unique pointer is invalidated, preventing any future accesses to that value.

To ease programming, unique pointers can be temporarily aliased, inside a lexical block, using a special alias pattern. An example of when this would be useful is when copying a pointer to step through an array. Copying a pointer would break the requirement to have a unique reference (unique means only one) so we therefore need to be able to (temporarily) alias a unique pointer.

```
void inc (int *‘r1 cell) {
  int *‘r1 t = cell;
  /*do something with cell */
  *cell = *t + 1;
}

void g() {
  int *‘U xptr = unew 3;
  { let alias <‘r2> int *‘r2 temp = xptr;
    inc (temp);
  }
  ufree (xptr);
}
```

Dynamic regions

Dynamic regions can be de-allocated at (almost) any program point. Hence when we dereference a pointer into a dynamic region, there is a runtime check that the region has not been already freed.

A dynamic region is therefore represented by both the state of the region—live or de-allocated—as well as the reference to the region itself. The state must remain after the region is de-allocated, since it may be consulted later.

To avoid redundant checking, Cyclone provides a lexical open construct, as the following example shows.

```
int *‘r bar(region_t<‘r> r, int * x)
{ return rnew(r) (*x + 42); }

void foo (dynregion_t<‘r,‘H> k)
{
  int *‘r x;
  { region h = open (k); //gives access to ‘r
    x = rmalloc (h, sizeof(int));
    *x = 42;
    bar(h,x);
  }
  free_dynregion(k); //destroys ‘r
}
```

Summary

- Garbage collection is good, but unsuitable for certain applications, most notably real-time applications.
- Regions are an experimental language feature for allowing a form of controllable automatic garbage collection, with small and predictable runtime costs. No programming language in widespread use currently supports regions.
- Regions are not currently the catch-all that one would like, and much research in the area of regions remains to be done.
- Regions can be supplemented with other forms of garbage collection, including dynamic GC, to give a fuller solution.