

# Garbage Collection Enable Perenniality

V. E. McHale

13 Aug. 2021

## Abstract

The choice of whether to include garbage collection in a language is also a choice of type system. Garbage collected languages correspond to classical, intuitionist logic and no language without garbage collection can properly support classical logic; perenniality divides logics where garbage collection divides languages.

Functional programming corresponds to intuitionistic logic via Curry-Howard; linear logic corresponds to functional programming without a garbage collector (Lafont 1988). The intuitionistic typing in garbage-collected languages features the perenniality characteristic of mathematical logic (Girard 2011, 331).

This perenniality is absolutely essential for a logic to have any flavor of mathematics, where proofs build up to theorems which are eternal truths; proofs make free use of previous theorems and theorems stand on their own: there is no obligation to consume results later.

## Programming Languages

We can see the flavor of “perenniality” concretely in programming languages. While linear logic departs substantially from mathematical logics, imperative programming and functional programming are mutually familiar.

## Linear Types

As a concrete example of how linear types allow us to work without a garbage collector, we can use ATS:

```
...  
{  
  val (pf | p) = bytearray_zero(40)  
  val () = bytearray_set(pf | p, 6, 1)
```

```

    val () = bytearray_free(pf | p)
  }
  ...

```

Had we tried to modify array contents after the byte array was freed, the program would be ill-typed and would fail to compile.

```

  ...
  {
    val (pf | p) = bytearray_zero(40)
    val () = bytearray_free(pf | p)
    val () = bytearray_set(pf | p, 6, 1)
  }
  ...

```

would be rejected by the compiler as `bytearray_set(pf | ...)` cannot be assigned a type because the type judgment for `pf` has already been consumed assigning a type to `bytearray_free(...)` (freeing a block of memory consumes the typing judgment).

Linearity also ensures that the program is memory-clean:

```

  ...
  {
    val (pf | p) = bytearray_zero(40)
    val () = bytearray_set(pf | p, 6, 1)
  }
  ...

```

would be ill-typed because `pf` is not consumed. The type of a heap-allocated value *cannot* be perennial: it tracks the lifetime of the value over the course of program execution.

## Perennial Type Systems

In Haskell one can write:

```

repeat :: a -> [a]
repeat x = let xs = x : xs in xs

take :: Int -> [a] -> [a]
take 0 _ = []
take n (x:xs) = x : take (n-1) xs

finiteList :: [Int]
finiteList = take 10 $ repeat 1

```

(`take` is not linear; `take 0 _ = []` discards one of its arguments)

Practically, this program works because of garbage collection; the typing is intuitionistic but calling `finiteList` will not leak memory despite abandoning values.

## Confused Type Systems

C has a **confused** type system: typing behaves like a perennially typed language, but does not fulfill its part of the promise. A value of type `float *` may be a pointer to an array of `floats` or it may be a pointer to garbage.

```
...
float* p = malloc(40);
free(p);
*p = 4.0;
...
```

will compile without any error despite being incoherent.

This is perhaps to be expected when typing imperative languages without Curry-Howard (Martini 2015).

## Conclusion

The Curry-Howard view divides languages by their approach to resource management and condemns those languages which use perennial type systems alongside manual memory management.

## ATS Headers

```
fn bytearray_zero {n:int} (int(n)) : [l:addr] (bytes_v(l, n) | ptr(l))

fn bytearray_set
  {l:addr}
  {n:int}
  { m : int | m <= n }
  (!bytes_v(l, n) | ptr(l), int(m), byte)
  : void

fn bytearray_free {l:addr}{n:int} (bytes_v(l, n) | ptr(l)) : void
```

In ATS, `!` means that the linear argument is not consumed.

## References

- Girard, Jean-Yves. 2011. *The Blind Spot: Lectures on Logic*. European Mathematical Society.
- Lafont, Y. 1988. “The Linear Abstract Machine.” *Theoretical Computer Science* 59 (1): 157–80. [https://doi.org/https://doi.org/10.1016/0304-3975\(88\)90100-4](https://doi.org/https://doi.org/10.1016/0304-3975(88)90100-4).
- Martini, Simone. 2015. “Several Types of Types in Programming Languages.” In *3rd International Conference on History and Philosophy of Computing (HaPoC)*, AICT-487:216–27. History and Philosophy of Computing. Pisa, Italy: Springer. [https://doi.org/10.1007/978-3-319-47286-7\\_15](https://doi.org/10.1007/978-3-319-47286-7_15).