Fast and Safe Memory Management

Giuseppe Maggiore Michele Bugliesi
Università Ca' Foscari Venezia
Dipartimento di Informatica
{maggiore,bugliesi}@dsi.unive.it

Abstract

In this paper we define a framework for manipulating memory through monads. We focus on a real-world class of applications, games and real-time simulations since these provide an excellent benchmark for heavy use of memory (lots of entities dynamically appearing and disappearing from the screen). Our objective is twofold:

- to make memory management simple and safe;
- to make memory management as fast as possible.

1 Introduction

Our goal is to define a system for managing memory safely and without leaks. We also strive for automated memory deallocation through the use of smart pointers rather than garbage collection for performance reasons. The kind of program that we will use as our main benchmark is a simple game where the player must avoid being hit by a certain number of asteroids. In this simple game our entities (the list of asteroids) are continuously allocated and deallocated:

```
type Vector2 = (Int,Int)
type Ship = Vector2
type Asteroid = Vector2

// library functions for accessing vectors
// ...

type State = Playing (Ship, [Asteroid], Int) | Won | Lost

// initialize the player ship
state := Playing(State(Ship(0,0), [], 0))

update dt =
  case state of
  | Playing(ship,asteroids,score) ->
    if Input.MoveLeft then
                ship.X := ship.X - dt
    if Input.MoveRight then
                ship.X := ship.X + dt
    foreach a in asteroids
      if a.Y > Screen.Height then
        asteroids.delete a
        score := score + dt
      if intersects a ship then
        state := Lose
      a.Y := a.Y + dt
    if asteroids.Count < CreateThreshold then
      asteroids.new(Asteroid(rand(0,Screen.Width),0))
    if score > MaxScore then
      state := Win
  | _ -> ()
```

The *update* function will be invoked very often; indeed, to achieve an interactive frame-rate for the simulation *update* must be called at least 20-30 times per second. Even more depending on how powerful the host CPU is. The *dt* parameter is a floating point value that represents how long it has been since the last invocation of the function and is used for numerically integrating the physics and AI equations of the simulation.

We will achieve our objective gradually. First we will give a formal definition of the source language and its type system. Then we will define a completely static system for managing memory that implements just a portion of our source language; this first system will be written in Haskell, which is an excellent tool for type-safe proofs of concept given its hugely expressive type system. Then we will extend this system with dynamic capabilities, at the same time migrating our code to the F# language which offers a more appropriate benchmark suite for performance. Also, given its vast set of real-world libraries such as XNA [XNA] for game development and given the language capabilities for implicit mutability allows us for some greater flexibility in the implementation of the internals of our system. Using F# comes at a cost: the language does not have the same powerful type system as Haskell does, so we will have to use explicit meta-programming features such as quotations [QUOTATIONS] and reflection [REFLECTION]. Finally we will add reference counting to remove the burden of freeing memory from the shoulders of the developer.

As a side note, our system offers two unintended, yet beneficial aspects. The first aspect is that memory is managed explicitly through a library; this makes some powerful extensions almost free to add. We discuss one of these that enables lock-less [LOCKLESS] concurrent programming by using the type of the state of mutable computations to regulate accesses. Various other possible extensions are discussed in [FUTURE WORK]. The second beneficial aspect is that all our entities fields are accessed through explicitly computed labels which can easily be passed around:

```
move_right entity position =
  entity.position.X := entity.position.X + 1
```

making it possible to implement self-describing objects [SELF REP] and some form of type-safe reflection.

Our main contributions are the following:

- to remove the need for a garbage collector in a high-level language to the great benefit of performance;
- to use phantom types to regulate memory accesses for safety and more.

2 Language

2.1 Language Definition