# PCF for game systems

Juan Casanova

August 31, 2019

## 1 Outline

The goal of the approach described in this document is to use simply typed lambda calculus with extensions (PCF [1] extended with polymorphism and type classes) as a basis for implementing systems in which we wish a high level of variability and combinatorial possibilities with as much complexity as possible. In order to do this, we need to consider what will be the types of our system, what features will be necessary and how they will relate to their game counterparts, keeping it as intuitive as possible, and potentially giving the players the ability to directly manipulate this system in some cases. Recursion and iteration are, of course, a must, through fix point operators and natural-number-like types. Polymorphism and type classes are quality of life improvements that largely improve the ability to manipulate the system itself. Finally, we discuss an approach to ensure termination and also control the magnitude of recursive elements by mimicking physics through an approach that we call *entropy-based evaluation.* This is necessary for real game implementations where we wish to give a large amount of control to the player or to randomized or semi-randomized systems in the game, as non-termination or high complexity growths could easily block the game from continuing its course properly, not to speak about its potential effects on game balance.

## 2 Simple type system

The basic underlying type system is the simply typed lambda calculus. The type system can be used for multiple systems within a game. For example, in Lambda rune we will use it for:

1. Spells

2. Maps / scenarios
3. Artificial Intelligence

The way in which it is used in each of this systems is that there is a final type that represents the output (a spell, a scenario, an AI script), and other auxiliary types that are used to produce this final types. For the remainder of this document, we will focus mostly on the spell use case of the system as an example.

The approach to spells in Lambda rune is tightly related to what we call the *effect system* that governs the gameplay of the game. A summary of this is that the game is conformed of *elements* that exist in the game (e.g. the character, monsters, spell props like fireballs or such, NPCs, obstacles, even the battlefield itself is an element), and nearly every aspect of the game is controlled through *effects* that go in, live within and go out of these elements. These effects are the core of the interaction, defining *local* (within each element in general, and on an effect by effect basis) rules for how some effects react to other effects, potentially changing and altering or producing other effects and elements. For instance, if a character casts a fireball to another character and damages it, the way this works is the following:

- An effect is added to the character casting the fireball that initiates the casting of the fireball.
- This effect creates a new element (the fireball projectile), and adds effects to it indicating its movement speed and its damaging nature.
- Once the fireball projectile collides with the target character, this produces effects on both the fireball projectile and the target character, indicating this event.
- On the fireball projectile, this simply produces the explosion of the fireball and its termination as an element.
- On the character, this produces a damage effect that goes into the character.
- The character has an ongoing life effect within it that reacts to the incoming damage effect, lowering the current hitpoints of the character and potentially killing it.

Within this system, the role of typed lambda expressions is to produce these effects, and as such, the final type for this system will be an *Effect* type. Once an element of this type is produced as the final result of an expression, it produces the casting of the corresponding spell in the character that cast it.

Formally speaking (see §4), Effect is a type class, not a type. Many specific types can implement this type class. For example, damaging effects, or moving effects, or life effects. Other type classes may exist to indicate effects that have a certain nature that makes them susceptible to certain manipulations. For example, effects with a duration might have their duration manipulated. Effects with damage might have their damage magnified. And effects that create an element might have effects added to said elements. These would all be type classes (DurationEffect, DamageEffect, ElementCreateEffect...).

Intermediate types also need to exist in the system, for the calculation of these effects. As a simple example, a type Damage may exist, indicating an amount of damage, which can then be used to create a DamageEffect, or alternatively, to create an OnDamageEffect that produces an effect whenever an element receives damage over a certain threshold. Another, perhaps more interesting, example would be to have a type ElementType which enables the identification of some elements based on their type, which can then be used for targetting purposes within many other different effects.

## 3 Polymorphism

Polymorphism in the sense of enabling forall-quantified type variables in the type system is fundamental for the flexibility that we wish to achieve in the system. Not only for definition purposes, but also for gameplay purposes. For example, in the spell system, our approach will be to provide atomic *gems* that implement atomic functions on the system, that can then be combined to create infinite amounts of more complex effects. In order to allow these gems to have interesting but also general enough effects, their types will need to be polymorphic in many cases. This, in combination with type classes (see §4) would allow things like an atomic gem duplicating the duration of a DurationEffect, or attaching an additional effect to every element created by an ElementCreateEffect. Without polymorphism, this would not be possible and individual gems would need to be created for each instance of DurationEffect and ElementCreateEffect, which would absolutely beat the purpose of the system.

## 4 Type classes

In close relationship with polymorphism, the ability to define type classes (traits) for types in the system is fundamental to be able to produce general functions that work for families of effects or other types. The multiple inheritance possible through type classes is particularly useful for having complex effects with plenty of aspects, each of which can be manipulated individually. As mentioned before, useful type classes may include things like DamageEffect, ElementCreateEffect or TargettingMethod.

## 5 Lambda abstraction

While it may seem obvious, an important aspect that needs to be present in the system is to be able to produce lambda abstractions of functions with variables. There is no particular details to be explained about this, but this ensures that the full computational power of the system can be used, and whichever front-end method we end up producing to generate the final lambda functions representing the effects, it needs to have the expressive power to include lambda abstractions within it.

## 6 Fix point combinator

We wish the resulting type system to be Turing complete to enable the full range of combinations and possibilities that this offers. In order to make simply typed lambda calculus Turing complete, a fix point combinator that permits recursion and a type that works like natural numbers (with successor, predecessor and iterating functions) are necessary. See §7 for a description of the latter. Here, we describe the former.

We will provide a general, pre-implemented, fix point combinator in the usual fashion. This will be a polymorphic function that can then be applied to any adequate function type. More precisely, it will be a function $\mathit{fix} :: (\alpha \to \alpha) \to \alpha$, for type variable $\alpha$. Thus, imagine that we have available a function that zaps a nearby enemy, and we wish to produce one that zaps *all* nearby enemies. We can do this by producing a function that targets the closest enemy, zaps it, then excludes it from the list of potential targets, then runs another function, and applying the fixpoint combinator to it.

Formally, we would have a function $\mathit{zapAndExclude} :: (\mathit{Context} \to \mathit{Effect}) \to \mathit{Context} \to \mathit{Effect}$ that checks the context, looking for the closest enemy, adds a zapping effect to the overall effect produced by the first parameter, changing the context to exclude said enemy.

Then, we can implement the element $zapAll :: Context \rightarrow Effect$ where $zapAll = fix\, zapAndExclude$.

## 7 Natural numbers

Another element necessary to achieve Turing completeness in a simply typed lambda calculus is a type that behaves like natural numbers, with sucessor, predecessor and some form of introspection function on the type. Our choice for this is to have an *iterate* function, of the form: $iterate :: Nat \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$. This can be, among others, used to implement a "check if zero" function, for example. Iteration of a function is likely going to be a useful tool in defining spells, for instance, and so this approach seems the most natural to our case. Because of this intention to ground the type into the semantics, we probably will not call it natural numbers, but instead something like Repetitions, making it explicit that it is used to produce Repetitions of functions. However, it will include addition and substraction functions to ensure the presence of successor and predecessor.

## 8 Ensuring termination: Entropy

A key aspect of all of this is termination. In the example of the spell system, effects are calculated *before* the spell is even actually cast. That is, this needs to happen in the least time possible (one frame in the game, for instance), and therefore termination needs to be guaranteed (and excessively long times avoided). But any Turing complete system cannot ensure termination, even less so high complexities. So how do we deal with this conundrum?

The reason why we want full Turing completeness is to be able to produce largely complex and unpredictable combinations that make the game harder to optimize and feel fresh more easily. True Turing completeness is not necessary. A way to maintain this level of complexity within boundaries, avoiding non-termination and very high complexities, and the one that we will follow, is what we call *entropy-based evaluation.*

The reason why we chose this name is that it mimics the notion in physics that entropy always increases within a system (if no energy enters the system). In our case it will be even simpler: all of our types will include an *energy* value, and all of our function types will include an *efficiency* value, such that when evaluating a function, the energy of the result is the energy of the input, reduced depending on the efficiency of the function. When the energy falls below a specific threshold, the evaluation instead defaults to a default value (where one is defined for each basic type within the system). The default value for function types $\alpha \to \beta$ will be a constant function that returns the default value for $\beta$. Because energy will always begin being a finite number, and efficiency will be a multiplicative value below 1, the value will always drop below any certain positive threshold, so termination will be ensured. Moreover, we can guarantee that as long as we do not provide any functions that have a growth which is larger than exponential, the result of applying it recursively with entropy-based evaluation will always have a result that is at most linear to the initial energy, enabling us to control magnitude as well as termination.

To provide an actual example, consider the example *zapAll* mentioned before. Imagine that as a side-effect of the effect, because of how it is produced, this effect also generates a new element in the map, within range to zap. Without the entropy system, this would produce an infinite loop that would result in the game freezing or crashing. However, with the entropy system, the result will be that a number of elements will be created, and all minus one will be zapped, but once the energy of the evaluation has run out after a certain number of runs (depending on the initial energy of the spell), evaluation will terminate. The physical intuition behind this system also makes a lot of sense in the context of the game, and potentially allows for much easier balancing and exploit controlling of the game: nothing is truly infinite.

## References

[1] G.D. Plotkin. Lcf considered as a programming language. *Theoretical Computer Science*, 5(3):223 – 255, 1977.