GAMMA: A Strict yet Fair Programming Language

Ben Caimano - blc2129@columbia.edu
Weiyuan Li - wl2453@columbia.edu
Matthew H Maycock - mhm2159@columbia.edu
Arthy Padma Anandhi Sundaram - as4304@columbia.edu

A Project for Programming Languages and Translators,
taught by Stephen Edwards

Why GAMMA? – The Core Concept

We propose to implement an elegant yet secure general purpose object-oriented programming language. Interesting features have been selected from the history of object-oriented programming and will be combined with the familiar ideas and style of modern languages.

GAMMA combines three disparate but equally important tenants:

1. Purely object-oriented

   GAMMA brings to the table a purely object oriented programming language where every type is modeled as an object–including the standard primitives. Integers, Strings, Arrays, and other types may be expressed in the standard fashion but are objects behind the scenes and can be treated as such.

2. Controllable

   GAMMA provides innate security by choosing object level access control as opposed to class level access specifiers. Private members of one object are inaccessible to other objects of the same type. Overloading is not allowed. No subclass can turn your functionality on its head.

3. Versatile

   GAMMA allows programmers to place "refinement methods" inside their code. Alone these methods do nothing, but may be defined by subclasses so as to extend functionality at certain important positions. Anonymous instantiation allows for extension of your classes in a quick easy fashion. Generic typing on method parameters allows for the same method to cover a variety of input types.

The Motivation Behind GAMMA

GAMMA is a reaction to the object-oriented languages before it. Obtuse syntax, flaws in security, and awkward implementations plague the average object-oriented language. GAMMA is intended as a step toward ease and comfort as an object-oriented programmer.

The first goal is to make an object-oriented language that is comfortable in its own skin. It should naturally lend itself to constructing API-layers and abstracting general models. It should serve the programmer towards their goal instead of exerting unnecessary effort through verbosity and awkwardness of structure.

The second goal is to make a language that is stable and controllable. The programmer in the lowest abstraction layer has control over how those higher may procede. Unexpected runtime behavior should be reduced through firmness of semantic structure and debugging should be a straight-forward process due to pure object and method nature of GAMMA.

GAMMA Feature Set

GAMMA will provide the following features:

- Universal objecthood
- Optional "refinement" functions to extend superclass functionality
- Anonymous class instantiation
- Static typing
- Access specifiers that respect object boundaries, not class boundaries

ray: The GAMMA Compiler

The compiler will proceed in two steps. First, the compiler will interpret the source containing possible syntactic shorthand into a file consisting only of the most concise and structurally sound GAMMA core. After this the compiler will transform general patterns into (hopefully portable) C code, and compile this to machine code with whatever compiler the user specifies.

Code Example

Example 1: Personhood

```
1  Class Person:
2    Protected:
3      ## instance variables have a type an access class
4      String first_name
5      String last_name
6
7      ## constructors take arguments, initialize state
8      init(String first_name, String last_name):
9        this.first_name = first_name
10       this.last_name = last_name
11
12   Public:
13     ## methods can return values and can be specialized
14     ## via refinement
15     String toString:
16       String result = this.first_name + " " + this.last_name + " is being stringified"
17
18       ## Only use this line if we have a defined
19       ## refining method
20       if refinable(extra):
21         result = result + " into a " + refine extra()
22
23       result = result + "!"
24
25       return result
26
27 Class Student extends Person:
28   ## subclasses have their own data
29   Private:
30     String level # Freshman, etc...
31
32   Protected:
33     ## Subclasses have to invoke their superclass,
34     ## just like in common OOP languages (java/etc)
35     init(string first, string last, string grade):
36       super(first, last)
37       this.level = grade
38
39   Public:
40     ## Now we can refine!
```

```
41      String toString.extra:
42        String level_result = this.last_name + ", " + this.first_name + " is a " + this.level
43        println(level_result)
44        return level_result
45
46  ## Call our class
47  Person average = new Person("John", "Smith")
48  println(average.toString)
49
50  println("")
51
52  Student very_smart = new Student("Roger", "Penrose", "Graduate Student")
53  println(very_smart.toString)
```

```
The above program should print:
    John Smith is being stringified!

    Penrose, Roger is a Graduate Student
    Roger Penrose is being stringified into Penrose, Roger is a Graduate Student!
```