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. Controlable

   GAMMA provides inanate security by chosing 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. Versitile

   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.

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 semmantic 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 with generic method parameter
- 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: The All-mighty Pattern

```
Class Person:
  ## constructors take arguments, initialize state
  _init(String first_name, String last_name):
    this.first_name = first_name
    this.last_name = last_name

  ## the run method is a fundamental method that makes an object callable
  _run:
    println(this.first_name + " " + this.last_name + " is being stringified!")
    result = refine
    println(this.first_name + " " + this.last_name + " is strinfigied into " + result + "!'
    return result

  ## Simple member objects as variables
  Protected:
    String first_name
    String last_name

  Public:
    ## This is a common pattern that is equivalent
    ## to declaring a method virtual / abstract
    ## we need to think about this more (necessary
    ## to implement via compiler check; maybe returns
    ## NULL object if not implemented, etc, etc...
    String citizenID:
      return refine


class Student extends Person:
  ## Subclasses have to invoke their superclass,
  ## just like in common OOP languages (java/etc)
  _init(string first, string last, string grade):
    ## We can interpret _init because it should only be called through the Person() constru
    ## Since it is called inside, it must refer to the super
    self._init(first, last)
    this.level = grade

  ## Now we can refine!
  _run:
    return this.last_name + ", " + this.first_name + " is a " + this.level

  ## subclasses have their own data
  Private:
    String level # Freshman, etc...

  Public:
    String citizenID:
```