Projeto Ouroboros

Sistema de integração automatizada entre C++ e linguagens de script

Nosso sistema visa auxiliar o desenvolvimento de aplicações em C++ que de alguma forma precisem interagir com uma ou mais linguagens de script – ou melhor, com as máquinas virtuais que as processam:

Para tanto, temos que satisfazer os dois sentidos da interação: a incorporação e a exportação. A primeira ocorre quando a aplicação quer acessar algo escrito nas linguagens de script, e o segundo ocorre na situação inversa.

Nesse trabalho, lidamos apenas com as linguagens de script Lua e Python.

Incorporação *(embedding)*

Suponha que você tenha um dos dois arquivos de script ao lado:

myscript.lua

```
variable = 42
function foo(x, y)
  return x+y
end
```

myscript.py

```
variable = 42
def foo(x, y):
  return x+y
```

E você queira acessá-los em sua aplicação C++ da seguinte maneira:

myprog.cxx

```
ScriptManager *manager = SCRIPT_MANAGER();
VirtualObj myscript = manager->LoadModule("myscript");
int script_integer = myscript["variable"].value<int>();
int result = myscript["foo"](9000, 1).value<int>();
```

Como podemos fazer isso? As máquinas virtuais de Lua e Python fornecem uma API em C para acessarmos os objetos criados via script, mas elas são bem diferentes:

incorporando com Lua

```
// L is a lua_State*
lua_getfield (L, LUA_GLOBALSINDEX, "variable");
// The value is put on the top of the stack
int value = lua_tointeger (L, -1);
```

incorporando com Python

```
// module is a PyObject*
PyObject * var = PyObject_GetAttrString (module, "variable");
// var contains the value we want
int value = PyInt_AsLong (var);
```

Para resolver isso, desenvolvemos uma API unificada chamada Ouroboros Project API (OPA). Ela usa polimorfismo com herança em orientação a objetos para abstrair a implementação para cada linguagem. Dessa forma, o usuário consegue lidar com objetos de script sem se preocupar com os detalhes de cada API, ou sequer saber se ele é proveniente da máquina virtual do Lua ou do Python.

Exportação

Por outro lado, se temos algumas definições em nossa aplicação, como:

myprog.h

```
class MyClass { /* ... */ };
int bar (int x) { return x+x; }
```

E queremos exportá-los para nossos scripts, dessa maneira:

myscript.lua

```
require 'myprog'
obj = myprog.MyClass ()
print ( myprog.bar(1337))
```

myscript.py

```
from myprog import MyClass
from myprog import bar
obj = MyClass ()
print (bar(1337))
```

Temos que adaptar as definições em C++ para que elas sejam compatíveis com as máquinas virtuais. Para tanto, colocamos uma nova definição "embrulhando" a antiga para que possa ser exportada adequadamente. E cada linguagem de script tem um protocolo próprio para isso:

lua_myprog_wrap.cxx

```
int wrap_bar (lua_State* L) {
  int arg0 = lua_tointeger(L, 1);
  lua_settop(L, 0);
  int result = bar(arg0);
  lua_pushinteger(L, result);
  return 1;
}
```

python_myprog_wrap.cxx

```
PyObject* wrap_bar(PyObject* self ,PyObject* args) {
  PyObject *arg0 = PyTuple_GetItem(args, 0);
  int arg0_value = PyInt_AsLong(arg0);
  int result_value = bar(arg0_value);
  PyObject *result = PyInt_FromLong(result_value);
  return result;
}
```

Mas esse processo é exaustivo. Então nosso sistema cuida disso através do Ouroboros Project Wrapper and Interface Generator (OPWIG), que analisa o código da aplicação para gerar proceduralmente os wrappers necessários.

Ferramentas

Dentre as tecnologias usadas no desenvolvimento, destacam-se o uso de C++11, os geradores de analisadores léxicos e sintáticos flexc++ e bisonc++ de Bob Brokken, o sistema de compilação cross-plataform CMake, a ferramenta de testes automatizados googletest e o serviço de integração contínua do Travis. Fora isso utilizamos, obviamente, as APIs de Lua e Python assim como, anteriormente, o gerador de wrappers SWIG.

Metodologia

Nós trabalhamos no projeto estabelecendo metas iterativamente que podem ser chamadas de milestones (marcos). Cada uma delas é uma aplicação simples que serve de teste para nosso sistema. Programamos o necessário para que a milestone atual funcione, e depois elaboramos a próxima – garantindo assim que nosso projeto atenda sistematicamente às características que lhe definimos, priorizando o que for mais fundamental e proporcionando um desenvolvimento incremental.

Referências e créditos

CMake. http://cmake.org/. Novembro 2013.
API Lua. http://www.lua.org/manual/5.1/manual.html#3. Novembro 2013.
Python. http://docs.python.org/2/c-api/. Novembro 2013.
SWIG. http://www.swig.org/. Novembro 2013.
Wikipedia: C++11. http://en.wikipedia.org/wiki/C++11. Novembro 2013.

Projeto de Fernando Omar Aluani e Wilson Kazuo Mizutani
(rewasvat@linux.ime.usp.br e wilsonkmizutani@linux.ime.usp.br),
com a orientação de Prof. Dr. Marco Dimas Gubitoso.

Trabalho de Conclusão de Curso, Novembro de 2013;
disponível em: *http://linux.ime.usp.br/~rewasvat/mac499/*
Bacharelado em Ciência da Computação
Instituto de Matemática e Estatística
Universidade de São Paulo