BabySteps: An approach to bootstrap an interactive system on COLA

Yoshiki Ohshima

This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

VPRI Memo M-2009-003

BabySteps: An approach to bootstrap an interactive system on COLA

Yoshiki Ohshima
yoshiki@vpri.org
January 4th, 2009

1 Introduction

In this memo, a path to bootstrap an interactive system on Pepsi is outlined. The Pepsi system requires the user code being translated to C. It hinders the interactive software development but more notably it is an obstacle to create a reflective system where the user code can query and manipulate the system itself.

At this point, we have a simple interpreter of the Pepsi language. The interpreter is mostly piggy-backed on top of the current Pepsi implementation that involves translation the code to C. The interpreter generates mostly linearized code (called SymCode designed by Ian Piumarta, which is analogous to bytecode but each instruction is represented as a symbol), and the symbol is used as a selector for the Context object. The execution of the code is done by sending the messages to the Context object.

We would like to have a highly reflective, self-modifiable whole system from this setting. In the next section, we describe what we would like to have. And in the following section, we describe how we would evolve the current system (without breaking too much along the line).

2 Goals

2.1 Performance Improvement

We are not deeply concerned with the performance of language, but it should perform reasonably. The current implementation is written in Pepsi and it involves dynamic dispatch at the Pepsi level. In other words, for each SymCode evaluation, there are dozens to hundreds of Pepsi message sendings required. In the end, it is about 1000 times slower than Squeak and Pepsi.

2.2 New Languages

The current Pepsi as a language is essentially Smalltalk-80 with some modifications, that includes 0-based collection indexing, additional unnamed arguments and varargs and etc. There are some baggages from the heritage and quirky aspects however; the collection library includes many specialized variations of similar functioning classes, and unifying "nil" and "false" objects and treating every other objects as "true" (to the author's opinion) complicates the language for users.

2.3 Objects Organization

Another thing we would like to explore is to have a highly loosely-coupled object system. We could explore it along the line of separated Object Tables. Having an Object Table allows us to reason about the objects and their relationships easily, and having more than one allows us to create a object reference graph that is better segregated (ala Croquet's islands).

An end-user authoring system built on top of this would let the system to put the user data and code into a dedicated Object Table, and saving it should be straightforward.

2.4 Code Organization

The SymCode compiler emits an array of symbols and basic values. We would like to store the resulting array in some form so that it can be used for introspection (in debugging etc.), and also can be used for faster re-compilation.

That is analogous to the image of Smalltalk, but we need to keep it in mind that the image should be "reconstructable" from the user code.

2.5 Reflective Environment and Tools

The system should be self-sustainable in real sense. It should know what it is doing, and it should be able to change any of its parts without breaking the rest. The user can then understand the system better with the help from the debugger and other development environment can take advantage of the reflective features.

2.6 Graphical User Interface and Graphics Engine

3 A Path to achieve these goals

A functioning debugger would be the first thing to write, and it should be written in the new interpreted setting. We have the Context objects but the compiler doesn't know the association with the textual code and the resulting SymCode. Emitting the source code info and storing them in the SymCode is the first step, and then a command line oriented debugger that doesn't throw away the chain of Contexts can be implemented easily. (Of course, later we would like to extend it).

We can generate an intermediate form for each compilation unit at the beginning. The reference to the global variables, etc. should be late-bound; upon loading such intermediate form, it should be resolved. (How to resolve is a question here.)

Making the new language should be able to done in parallel and gradually. The new language should almost look like Pepsi, but we would add a new name space for the new types in the new language. Gradually, we define a new type (say, in the collection library), and instead of clobber the definition in "objects.a" that is linked to the interpreter, we make the new definitions live in their own world. (Later on, we would of course try to merge them.) The debugger wouldn't be able to look into the C-compiled Pepsi code, but the migration allows it gradually usable for more code.

The performance improvement should be made in parallel with above. At the beginning, we could eliminate the dynamic message sending in the implementation of code by generating Coke code (or C) ala Slang. But this is just for the time being, hopefully. Eventually, we would like to have some JIT. Ian's effort of making dynamic Pepsi, or some other ways of making JIT would be good.

The ideas around the object tables may take some time to be materialized. In the meantime, we would write the special objects like process scheduler (that deals with multiple chains of Contexts) in a way so that it interacts with the rest of system only via message sendings. Later, we would refine it so that swapping scheduler dynamically would be possible.