Functional Scripting

a study of discipline and implementation for operations of external resources in functional language

YAMATODANI Kiyoshi (010119)

School of Information Science,
Japan Advanced Institute of Science and Technology

February 15, 2002

Keywords: functional programming, scripting, component, database, polymorphic type system.

To cope with the complexity in development of large software, a "component-based" method is emerging. In this method, applications are built by connecting components. This innovated method requires new languages that enable the programmer to assembly components in a systematic way. For this reason, scripting languages are attracting attentions. They provide powerful foreign function interfaces which makes them 'glue' of libraries and modules. While this approach works reasonable well for small programs, it does not scale up well to complex and large software systems. The untyped nature of those scripting languages makes it difficult to achieve the reliability and robustness required for large scale software.

A well established method for enhancing software reliability is to impose static type discipline. Moreover, modern functional languages with advanced static type system also provide high descriptive power suitable for rapid application development. The underlying theory of functional language however implicitly assumes that the runtime system is self-contained and does not provide any mechanism for interfacing with foreign components. There are some recent researches on foreign language interfaces for functional languages, but they remain rather low-level and cannot provide the power and flexibility comparable to those of scripting languages.

The motivation of this thesis is to establish theory and implementation techniques for combining the safetiness of functional languages and the openness of scripting languages. To achieve this, we must establish following

- a typing system for external access,
- a method to integrate object oriented type system and a static polymorphic type system,
- a programming language with the above type system and interfacing scheme for external libraries,
- an implementation method for the language including compilation scheme and runtime mechanism for external resources.

This thesis attempts to provide solution to the above problems.

I have developed a static polymorphic type system for external access by introducing 'object type' which represents properties of external object. An object type is a form of abstract type hiding its physical structure, but allows pattern-matching with record patterns . I have carried out the formal development of the type system based on the polymorphic record calculus by Ohori. In the resulting type system, functional programs can import objects exported by external libraries without losing their property by assigning the object type to those objects.

I have developed a method to encode object oriented class hierarchy in the type system. The subtype relation in an OO type system is mapped to the combination of object type with record polymorphism. In this scheme, I have shown that the constraints which OO type system imposes are preserved by this mapping. This method is more natural and portable than those developed by other studies which try to merge OO type system and functional type system.

I have designed an ML-like language 'Amethyst ' which embodies above theoretical development. Amethyst supports core syntax of Standard ML, and adds some declaration statements for importing external resources. Type system checks operations on values imported by these statements and guarantees that assumptions expected by external libraries will be satisfied. In addition to the language, I have also designed an abstraction layer called 'domain module' between the language runtime system and external libraries. Domain modules hide implementation details of external libraries and re-construct abstract interfaces which reflect original data-model of these libraries.

Following above base design, I have implemented Amethyst system which consists of three subsystems : a compiler, a bytecode interpreter and domain modules. The compiler performs type inference with external type and translates source programs which operate values of external types to bytecodes which invoke functions implemented in external libraries. The bytecode interpreter is an extension of Leroy's ZINC machine with new instructions for operating external data and for invoking external functions. The interpreter also incorporates heap management system designed for storing the information related with external resources. Lastly, we designed and implemented two domain modules. One provides interface to PostgreSQL database, and the other provides interface to Java class library. These modules demonstrate that our approach can be applied to development of real applications.