Call-by-Value Semantics for Mutually Recursive First-Class Modules

Judith Rohloff

Modularisation in an essential tool for managing complexity in the design of large scale software. To be effective, module systems must organise code into fine grained hierarchies without impeding reusability. The competing nature of these goals has given rise to several approaches for designing module systems. With parametric modules, adaptable code that is easy to reuse at different parts of the architecture. First-class modules [1,2,3], i.e. modules that can be arguments and results of functions, allow parametric modules to be defined as normal functions, so no special construct is needed. Furthermore, first-class modules enable runtime reconfiguration of the architecture at runtime. With nested modules [4] a fine grained module hierarchy is possible. Disallowing mutually recursive modules [5,6,7] often destroys the natural structure of a program.

Recent approaches [8,7,9] have attempted to combine all three features in a call-by-value language, but none of them have achieved full support. In this presentation we describe a way to give a call-by-value semantics to a language with higher-order functions and nested, first-class, mutually recursive modules.

We use the grouping concept proposed in [10] as modules and call the proposed construct *module*. Pepper introduces the concept of *modules* as a unified construct for bindings and data structures. *Modules* are sets of definitions. They are first-class values which capture recursive definitions, lexical scoping, hierarchical structuring of programs and dynamically typed data structures in a single construction. As first-class values, they can also be used as records. They distinguish from records in that all definitions within a module can see each other. In [10] *modules* are treated coalgebraically and regarded as objects implicitly defined by their observers.

In contrast to this semantic approach, we focus on an efficient implementation by defining formal semantics and analyses for *modules*. For this purpose we define a small functional call-by-value language ModLang, which is similar to an untyped λ-calculus extended by *modules*.

In a call-by-value semantics, an evaluation order is needed, as every variable must be bound to a value before it is used. There are two possibilities to obtain this order. Either the programmer has to state all definitions in dependency order (e.g. ML), or the order is determined via dependency analysis (e.g. Opal or Modula-3 [11]). As *modules* are sets of definitions, no order is given. As *modules* are first-class values, have binding definitions and are allowed to be mutually recursive, it is not easy to find the evaluation order. On the one hand we have to look over *module* borders and on the other hand path resolution is undecidable as proven in [8] so it is not possible to detect the corresponding definition for all selections.

Therefore, we developed a special dependency analysis for our language. The result of the analysis is used to transform the input program into an intermediate representation with explicit dependency order. For this intermediate representation we have defined a denotational semantics. Finally, we have already extended our language ModLang by *module* morphisms. They allow *modules* to not only be created at runtime, but also to be extended or restricted. This improves the flexibility and reusability of *modules*. Furthermore, we have a mechanism for imports based on a control flow analysis and the ability to control visibility when using imports and exports.

References

1. Peyton Jones, S.L., Shields, M.B.: First class modules for Haskell. In: 9th International Conference on Foundations of Object-Oriented Languages (FOOL 9), Portland, Oregon. (January 2002) 28–40
2. Russo, C.V.: First-class structures for standard ml. Nordic J. of Computing **7** (December 2000) 348–374
3. Reinke, C.: Functions, Frames, and Interactions – completing a lambda-calculus-based purely functional language with respect to programming-in-the-large and interactions with runtime environments. PhD thesis, Universität Kiel (1997)
4. Blume, M.: Hierarchical Modularity and Intermodule Optimization. PhD thesis, Princeton University (November 1997)
5. Crary, K., Harper, R., Puri, S.: What is a recursive module? SIGPLAN Not. **34**(5) (May 1999) 50–63
6. Nakata, K., Garrigue, J.: Recursive modules for programming. SIGPLAN Not. **41** (September 2006) 74–86
7. Russo, C.V.: Recursive structures for standard ml. SIGPLAN Not. **36**(10) (October 2001) 50–61
8. Nakata, K., Garrigue, J.: Path resolution for nested recursive modules. Higher-Order and Symbolic Computation (May 2012)
9. Odersky, M., Cremet, V., Röckl, C., Zenger, M.: A nominal theory of objects with dependent types. In: Proc. ECOOP'03. Springer LNCS (2003)
10. Pepper, P., Hofstedt, P.: Funktionale Programmierung – Sprachdesign und Programmiertechnik. Springer (2006)
11. Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., Nelson, G.: Modula-3 report (revised). ACM SIGPLAN Notices **27**(8) (1992) 15–42