Module Transformations in Parameterized Programming

Uragami Takahiro

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

14th Feb. 1997

Keywords: parameterized programming, reusability, module graph, CafeOBJ.

1 Introduction

Structuring is the indispensable method for developing any large-scale and/or complicated software, so a mechanism for modularization is important. It is also important in specifications written in natural languages. Usually specifications written in natural language have some structures such as chapters and sections. Also, table of contents and indexes are attached to specifications. These make hierarchical structure of specifications by the reference relationship. This relationship decides whole structure of specifications.

When you read a specification written in natural language and find an outline with indexes to detailed definition parts, it makes-up a hierarchical structure. If you find an application of a function and there is an index to the detailed definition of the function, we can say it makes-up a reliance relationship. Further more, the reusability of specification is improved by using function parameterization. In natural language specifications, parameterization is expressed in the declaration of that the functions rely on implementation.

In algebraic specification paradigms, a set of data and operations are all abstracted and their properties are self-supported what is called data abstraction. Abstract data type is typical method for modularization. The mechanism for modularization should have methods for referring each other such as importing other modules. It makes use of reference relationship and parameterized specification. These mechanisms make it easy to develop complicated reference relationship between modules, but it may lose module readability. We propose a method called module graph to increase the readability. Which makes it easy to understand module relationship. A module graph is based on the same concept with the table of contents and the indexes of specifications in natural language. After defining the module graphs, we try to show the effectiveness of this method.

2 Algebraic Specification Language CafeOBJ

CafeOBJ is an algebraic specification language based on order-sorted rewriting logic which is an extension of order-sorted equational logic. A subset of CafeOBJ is executable, where the operational semantics is given by a conditional order-sorted term rewriting system. In CafeOBJ, we can use parameterized programming mechanism that is inherited from OBJ2 and OBJ3. For example, we can define a parameterized stack module STACK in CafeOBJ. The parameter of STACK is characterized by the theory TRIV. TRIV is the requirement theory for an interface that only requires a designated sort corresponding to Elt from an actual object. Using this generic STACK object we can define integer stack INT-STACK only by instantiating TRIV with INT. As this, parameterized modules maximize reusability by permitting tuning to fit a variety of applications. Also it increase the expressive power of modules. For example, a parameterized sequence module can be imported into a sorting module without instantiating. This module will be a sorting program over sequences of any sort. At the time, the sorting module is parameterized with the same parameter as sequence module. When the sorting module is instantiated with any module, the same module will bound the parameter of the sequence module. We call this sequence sorting module as a multiple parameterized module. We call this event as a multiple instantiation.

Using CafeOBJ's renaming mechanism, we can import one module twice with different name. Multiple parameterize mechanisms and renaming functions are very useful properties for programmer, because both of them maximize module power of expression. Unfortunately, unlimited using of these properties makes program code more complicated and lose module readability.

3 Module Graph

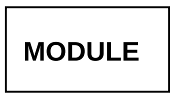
A module graph is heavily related to the information contained in header part of the CafeOBJ module. This header part contains a module name, of its self importing module name and its mode, parameter and constraint module name, and renaming informations. The declaration except module name are optional. A module graph denotes the structure of modules. Its nodes denote modules (figure 1) and its edges denote relationships between imported modules and importing modules (figure 2). Parameterized modules are represented with parameters denoted within an oval (figure 3). Module shapes will change before and after instantiating. This variation makes it possible to represent a specification which is in an intermediate stage. An instance of a parameterized module is given by binding modules to parameters. The process of binding is called instantiation. Instantiation of parameters can be done by declaring a view with the module graph. If the view already declared then, their name is bound to the formal parameter. If there is no view, you can declare an ephemeral view in the module definition and use it. Views are represented as diamond with a constraint and an actual module (figure 4). View is put in the diamond and arrows are draw from a line which represents the bind of the target module to the constraint module. Instantiation mechanism is expressed as an edge directed from view to formal parameter, which represents the parameter of parameterized module graph (figure 5). The multiple parameterization will shape as parameterized modules are inside of a parameterized module (figure 6). The parameter part of these modules are connected with arrows.

4 Conclusion

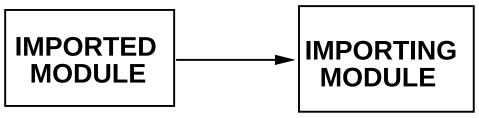
Using our method we can express multiple parameterized modules and modules with renaming mechanism as the module graph. These structures play important role for modularizations and readability. We can easily understand the relationships of module references from the module graph. Further more, we can add or improve modules using its the module graph. In this paper we show some examples consist of data structures by module graphs. Then, we changed the data structure of the module on the graph keeping the module structure unchanged. Possibility of improving a program without changing module structures means that we can safely maintain the program.

As a result, we find that the change of the data structures on a module graph shows us the changes of constraints for parameters. In the multiple parameterized module, if the constraint of inner module is more tight than that of outer one, the module definition regarded as invalid, so this case we have to change the constraint of the outer module to be the same as inner module. This shows that the module graph shows not only the state changes of module structures but also shows the changes of the constraints for the formal parameters.

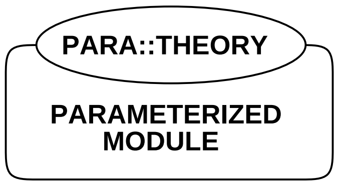
Module Graph Definition



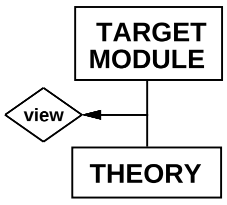
(1) module



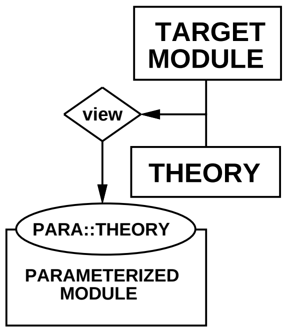
(2) module importing



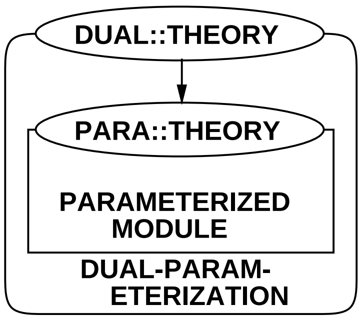
(3) parameterized module



(4) view



(5) instantiation



(6) dual-parameterized module