Available online at www.sciencedirect.com

SciVerse ScienceDirect

Physics Procedia 33 (2012) 1657 – 1662

Physics Procedia

2012 International Conference on Medical Physics and Biomedical Engineering

Automatic Generation from List comprehension to C

Qiushuang Wang[1,a], Dongliang Liu[2,b1], Rujuan Wang[3,c], Yan Sun[4,d]

[1]*JiLin University*
5333, XiAn Road, ChangChun City
JiLin Province,130012,P.R.China
[2]*JiLin University*
5372, Nanhu Road, ChangChun City
JiLin Province,130062,P.R.China
[3]*College of Computer Science and Technology*
Jilin University
Changchun 130012, China
[4]*College of Computer Science*
The city college of jilin architectural and civil engineering institute
changchun 130000,China
[a]*25501120@qq.com*, [b]*125258889@qq.com*, [c]*wangrujuan_1108@sina.com*, [d]*sunyannenu@163.com*

Abstract

List Comprehension representing comes from comprehension of mathematics, which provides a strong and concise representing method. Monad representing is concise and readability mainly because Monad extends List Comprehension to any Monad. This paper is concerned with the transforming from List Comprehension of functional programming language Haskell to C programs. Using Haskell itself we have also realized this transformation. Haskell is the only language that supports Monad, so our research is helpful to the building of Monad researching environments.

Keywords: List Comprehension; Monad; Haskell; Transforming.

1. Introduction

[1] Correspondence: Dongliang Liu, JiLin University, 5372 Nanhu Road, ChangChun City JiLin Province, 130012, P.R.China, Tel:13596040093, Email:125258889 @qq.com

 doi:10.1016/j.phpro.2012.05.267

As a new formal description method, Monad emerges in recent years and it is a new technology that has important theoretical value and extensive application prospect. The emergence of Monad makes the application of category theory in computer science comes into a reality, which has been treated as a generic programming technique by experts [1]. List Comprehension is a conversion of collection mathematics comprehension, a collected comprehension realizes the collection through the definition of each element, which is a very powerful and simple description.

Professor Philip Wadler in University of Glasgow induced the simplified representation into Monad [2] in 1992, and gave out the comprehension definition of Monad. Generally speaking, a comprehension owns the form of [t | q], among which t is a term and q is a qualifier or empty Λ; or a generator x←u (where x is a variable and u is a term whose value is list); or the compound of qualifier (p, q). Comprehension is defined by the following rules:

[t | Λ] = unit t

[t | x←u] = map (λx→ t) u

[t | (p, q)] = join [[t | q] | p]

Among which we assume t, u, v as representative items, and p, q, r is on behalf of qualifiers. Unit, map and join are the three polymorphic functions. Suppose L1 and L2 as the list and with comprehension:

$[(x,y)| x \leftarrow L1, y \leftarrow L2]^{List}$

The comprehension is expressed by the new list constructed by list L1 and L2. Specifically speaking, if L1 = [1,2], L2= [3,4,5], thus the new list of the comprehension definition is [(1,3), (1,4), (1,5), (2,3), (2,4), (2,5)]. The conciseness and readability of Monad representation is due to that the Monad has promoted L comprehension representation to any Monad, which is a major contribution to the Monad.

Currently Monad is still at the exploratory stage, which focuses on theoretical research and ignores practical application. Its large number of technology is not mature and some confirmations are theoretical rather than being received through a large number of application. The key point of Monad application research is to establish a supporting environment for Monad, as the issues that have been researched and confirmed must be examined and tested through application examples. Nowadays the programming system that supports Monad methods is only the sub-system of Haskell. This paper considers List Comprehension expression of Haskell language as the study object, proposes the automatic generation method from List comprehension to C, and finally achieves this conversion with Haskell language, among which the functional languages is selected to develop the model rapidly rather than being sank into the details of the process of solving problems. Comprehension is an important defined form of Monad and the research on its automatic generation is helpful to build the Monad implement system, which can provide supporting environment for the application research of Monad.

2. The definition of List Comprehension in Haskell

One of the features in Haskell language is the introduction of List Comprehension representation, which defines a list through the expression that each element in the given list meets.

2.1 List Comprehension syntax

List Comprehension syntax in Haskell is defined as follows:

ListComp → [exp | qual1, ⋯ , qualn]

Qual → pat ← exp | let decls | exp

The basic form of List Comprehension is: [e | q1, q2, ..., qn] ($n \geq 1$), where qi is the constraint. A List Comprehension returns a list, which is obtained by calculating objective expression e in the environment of generating constraints. The calculation of constraints is done according to depth-first rule.

B.Semantics of List Comprehension

The informal semantics of List Comprehension can be described by the following equation:

[e |] = [e]
[e | b , Q] = if b then [e|Q] else []
[e | p l , Q] = let {ok p=[e|Q]; ok _=[]} in concatMap ok l
[e | let decls , Q] = let decls in [e|Q]

Where e is the expression, p is the mode, l is a list expression, b is a Boolean expression, decls is to declare the list, Q is the binding sequence and function concatMap is the function to realize the system predefinition in Haskell.

3. The Conversion method of List Comprehension

List Comprehension is a kind of higher-order structure, which owns the high level abstraction and simple description for the calculation process to avoid the recursive definition. In the process of converting into C, we need to convert higher order into first order.

A representation form that is similar with List Comprehension proposed in literature——Iterated Comprehension [3] is to convert each constraint into a recursive function and then use the value of constrained variables to call this recursive function, thus n constraints qual1,…, qualn are converted into the recursions with N nests. This method involves in the definition of N new local functions and needs to be converted into the definition of global function through λ promotion. In addition, the operation of recursive function consumes a lot (its memory consumption is O (n) and n is the height of the tree that to be called), so in order to improve efficiency, we need to convert recursive function into cyclic iterative structure (its memory consumption is O(1)） . The List Comprehension is the concise representation in syntax, which has combined the high-order functions in Haskell such as Concat, Map, and Filter.

A conversion method can first convert List Comprehension into the combination of higher order functions, and then transform higher-order function into C, but the conversion from higher-order function to the first order function also requires complex instantiation process [4].

According to the semantic analysis of List Comprehension, this paper converts List Comprehension to the cycle nested structure of C by scanning just for one time.

4. Conversion algorithm

List Comprehension is an expression in Haskell and the conversion is actually to convert the abstract definition of the expression into specific Solution of C. Before the conversion, the system is faced with a text form of the Haskell source. Therefore, it first needs to convert the source program into an internal representation - abstract syntax tree and then converts the internal representation into C. The conversion results are made up by five components: variable definition section, the statement sequence, expression, the expression result type and type definition.

4.1 Conversion of Data type

The calculation results of List Comprehension are list types in Haskell and list types can be converted to the list structure of C, for example: the structure of integra list [Int] corresponding to C is:

```
        struct List
{  int elem;
   struct List *next;
     };
```

Operations on the types of list: elements plus list, the list pluses the list, take the header, take the footer, etc. are respectively converted into corresponding C-list operation.

4.2 Conversion of list constants

Exp in Generator x ← exp may be the list constant. When converting list constants, because there is no corresponding constant definition in C Language, we can only first define the structure type corresponding with the list , then declare a variable of this type, such as: struct List * alist; and finally achieve by the method of assigning value to this variable.

4.3 Convert the variable

Exp in Generator x ← exp may be a variable of list types. There are no changeable variables in Haskell language and you can bind a name with a value, but once bound, the name only represents that value within its scope, not allowing any changes. This is different from the definition and assignment of C variables. C variables such as "counter" or "container", you can assign a value to a variable and then another value to the variable. Therefore, in the generated C codes, there are no statements that can change the "variables" in Haskell and if you want to make changes, you must first declare an equivalent variable and then operate the newly defined variables.

4.4 Conversion Algorithm

The main algorithm that converts the internal representation of List Comprehension into C is as follows:

```
tListcomp [Listcomp exp quals] <env><tir>=
let  (v,s1,s2,e,t,tir1)=TransQuals [exp quals] < env><tir>
in  (v, s1++s2, e, t, tir1)
```

The statements generated by List Comprehension may include statements established by linked list, which needs to apply for memory space for each element of the linked list. This part of code is a little longer, so for the clarity, we should store the statements to establish linked list（s1）and other statements（s2）separately. After the conversation, merge them together (s1 + + s2) to obtain the final sequence of statements.

As the constraints in List Comprehension may be nested (the latter utilizes the variable bound by the former constraint), we need to establish an environment list Env in the process of conversion:

```
tLastQual [defide←val] <env><tir>=
let  (v1,s1,valide ,tyname,tir1)=transExp [val] <env><tir> 1*
   valide'=newide ()                                          2*
   ty=getBasety [tyname] <tir1>                              3*
   (dv,ds,de,dty,tir2)=transExp [exp] < (defide,ty):env ><tir1>                                  4*
   rty'=gettydef dty tir2                                     5*
rty=if rty'==" " then newide () else rty' in
rtir=if rty'==" " then (Tseq rty dty): tir2 else tir2  in
(v1++dv++"tyname valide'; ty defide; rty ride, newide, tempide; int count;",
s1++"ride=NULL; count=0;", "valide'=valide;
while (valide'!=NULL){defide=valide'    elem; ds;
If (count==0){newide=tempide=(rty) malloc (LEN);
newide    elem=de; newide    next=NULL; ride=newide;
```

else {newide=(rty) malloc (LEN); newide elem=de;
newide next=NULL; tempide next=newide; tempide=newide;}
count=count +1;valide'=valide' next;}", ride, rty, rtir)
Type Env=[(String,String)]

The sequence of a unit doublet (identifier, type) in environment list, The constraint is calculated by the depth-first rule, if the constraint is the generator: x←exp, then add the x and its type into the environmental list; if the constraint is locally defined: let decls, thus the local variable declared in the decls and its types will be added into environment list. After each treatment of constraint, the environmental list will continue to accumulate, besides, the process on the latter constraint will use the environment generated by the former constraint. After processing the final constraint, the target expression will be formed in the final environment list. The environmental list will be withdrawn when the List Comprehension processing is terminated.

The conversion algorithm of the constraint part is a little longer and the following is to explain the code generation, taking--the final constraint is generator defide←va -as an example. (Assuming the List Comprehension that the constraint lies in is defined as: [exp|qual1,…, defide val])

Description of the algorithm is as follows:

1) TransExp function is used to convert the expression val in the generator defide←val as the C code.

2) As there is assignment operation in C code corresponding to generator defide←val, so we need to define a new variable valide' and the generated C statement can conduct this new defined identifier operation.

3) GetBasety function can obtain the base type of the expression val from the type definition list, which is used to define the type of variable defide.

4) Add the variable defied defined by the final constraint and its type into the environmental list. The target expression of List Comprehension utilizes the environmental list formed by the accumulation of all the constraints to generate the target code.

5) According to type defined list, gettydef function will judge whether the type of list that considers the type of target expression as the basic type is defined, if it is not fined, gettydef returns to an empty string, at this time, the definition of such kind of list type should be added.

5. Conclusion

This paper presents the automatic generation method of List Comprehension in functional language Haskel to C. List Comprehension is the main definition form of Monad. Monad owns not only good computing infrastructure of abstract categories, reflectivity and reusability, but also the merits of being apt to automatically realize expansion and modification. It will bring a new concept, new method and new technology to computer software, which is expected to be widely used. The realization of List Comprehension is helpful to build a supporting environment for the applied research of Monad, we have used Haskell language to achieve the automatic conversion system, which indicates that the conversion method in this paper is feasible.

References

[1] Philip Wadler. The essence of functional programming [R]. In Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages, pp. 1-14, Albuquerque, New Mexico, January 1992.

[2] Philip Wadler. Comprehending Monads[J]. Mathematical Structures in Computer Science, 1992,2(3):461~493.

[3] Wolfgang Schreiner. Compiling a Functional Language to Efficient SACLIB C*[R]. Technical Report 93-49, RISC-Linz, Johannes Kepler University, Linz, Austria,1993,9.

[4] George Horatiu Botorog, Herbert Kuchen. Translation by Instantiation: Integrating Functional Features into an Imperative Language[R]. In:Proceedings of the Poster Session of CC'96, Research Report LiTH-IDA-R-96-12, University of Linkoping. 1996.

[5] Filinski A. Representation Monads[R]. In: Wing JM, ed. Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Oregon: ACM Press, 1994. 446~457.

[6] Simon Peyton Jones. Haskell 98 Language and Libraries[M].Cambridge University Press,2003.

[7] Simon Peyton Jones, John Hughes. Report on the Programming Language Haskell 98, A Non-strict Purely Functional Language[R]. Yale University, Department of Computer Science YALEU/DCS/RR-1006,1999.

[8] A Santos.Compilation by transformation in non-strict functional languages[Ph.D. Thesis].Department of Computing Science, Glasgow University,1995.

[9] Simon Peyton Jones. Tracking the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell[C]. In: Tony Hoare, Manfred Broy, Ralf Steinbruggen eds. Engineering theories of software construction, IOS Press, ISBN 1 58603 1724, 2001: 47~96.

[10] John Hughes. Generalizing monads to arrows [J]. Science of Computer Programming,2000:37:67~111.

[11] Tim Sheard, James Hook. Fine Control of Demand in Haskell[C]. In: Proceeding of Sixth International Conference on Mathematics of Program Construction,2002.