韓國 OA學會 論文誌
第3卷 第1號, 1998. 3.

論文 98-3-1-1-3

Annotation을 가지는 함수언어의 설계 및 번역기 전반부 구현

최 관 덕*

The Design of a Functional Language which has an Annotation Syntax and Implmentation of the Front-end of the Translator for the Language

Kwan-Deok, Choi*

요 약

함수 언어에서 병렬성을 표현하는 주된 기법으로는 스트릭트니스 분석과 annotation이 있다. 스트릭트니스 분석은 번역기가 병렬성 탐색을 수행하여 탐색된 정보를 목적 프로그램에 표현하는 기법이며, annotation은 프로그래머에게 병렬성 탐색을 맡겨서 원시프로그램에 표현하도록 하는 기법이다. 본 연구는 annotation에 관한 것으로 annotation 구문을 가지는 원시함수 언어와 이를 위한 번역기의 전반부를 설계하고 구현하는 것을 목적으로 한다. 번역기의 전반부는 원시함수언어 프로그램을 중간 언어인 확장 람다계산 그래프로 번역한다. 번역기는 UNIX 환경에서 컴파일러 자동화 도구인 YACC, Lex를 사용하여 C로 구현한다. 본 논문에서는 번역기에 사용된 구현기법에 대해서 기술한다.

Abstract

There are two major method for expressing parallelim in functional languages. The one is the strictness analysis and the other the annotation. The strictness analysis is a method that a compiler detects parallelism and expresses the detected information in the object program. The annotation is a method that a programmer detects parallelism and expresses in the source program. This study is on the annotation and is aimed at construction of a translator for a functional language which has an annotation syntax. The translator translates a source program to enriched lambda-calculus graphs. The translator is implemented in C using compiler development tools such as YACC and Lex, under UNIX environments. In this paper we present the design and implementation techniques for developing the front-end of the translator.

---

* 대구전문대학 전자계산과 전임강사

논문접수 : 98.1.5. 심사완료 : 98.2.25.

Ⅰ. 서 론

함수 언어는 수학적 함수 개념을 프로그래밍 언어화한 것이다. 함수 언어의 가장 큰 특징은 대입문에 의한 부작용이 없어서 참조 투명성(referential transparency)을 보장한다는 것이다. 이 특징은 실행이 종료한다면 실행 순서에 관계없이 실행 결과가 일정하다는 성질을 갖도록 해주기 때문에 함수 언어는 식의 검증과 병렬 처리에 유용하다 [1,2].

함수 언어의 병렬 처리 모델로는 평가 가능한 식을 모두 병렬로 계산하여 필요한 것만을 사용하는 투기적 병렬성(speculative parallelism)을 이용하는 방법과 함수의 평가에 꼭 필요한 식만을 계산하는 보수적 병렬성(conservative parallelism)을 이용하는 방법이 있다. 투기적 병렬 평가는 필요하지 않은 인수를 평가할 가능성이 있을 뿐 아니라 그로 인하여 식의 종료 특성이 변경될 수도 있다. 따라서 보수적 병렬 평가를 병렬 처리 모델로 선택하는 것이 바람직한데, 보수적 병렬 평가를 지원하기 위해서 순차 처리 모델에서의 지연 평가(lazy evaluation)를 사용하는 경우에 병렬성은 스트릭트한 기본함수로 국한되기 때문에 병렬성을 증대하기 위한 기법이 필요하게 된다. 병렬성의 증대를 위한 방법으로는 꼭 평가될 식을 프로그래머가 프로그램 상에 명시적으로 annotation 하는 방법과 컴파일러가 프로그램을 분석하여 알아내는 방법이 있는데, 후자를 스트릭트 니스 분석(strictness analysis)이라 한다 [3].

본 연구는 함수 언어의 병렬성 증대를 위한 기법 중에 annotation에 관한 것이다. 이를 위해서 annotation 구문을 가지는 원시 함수 언어를 설계하고, 그것의 번역기 전반부를 설계하고 구현한다.

Ⅱ. Annotation

함수 프로그램을 병렬 실행하는 경우에 고려할 것은 크게 두 가지로 요약될 수 있다. 하나는 식의 어느 부분을 병렬로 실행해도 안전한가 하는 것이고, 또 하나는 식의 어느 부분을 병렬로 실행하면 효율적인가 하는 것이다. 이 두 가지에 대한 정보는 프로그래머나 번역기에 의해서 분석이 되어서 표현되어야 한다. 이때 이러한 분석을 병렬성 탐색이라 하며, 분석 결과를 표현하는 것을 병렬성 표현이라 한다.

병렬성 탐색을 번역시에 스트릭트 니스 분석을 통하여 자동적으로 하게 되면 프로그래머는 병렬성에 대한 것을 신경 쓰지 않고 프로그램을 작성할 수 있다. 그러나 스트릭트 니스 분석이 지연 함수 언어의 병렬성 탐색을 위한 실용적인 기법으로 사용되는데는 아직은 문제점이 있다. 그 문제점은 스트릭트 니스 분석이 완전한 병렬성 정보를 추

론할 수는 없다는 점과 번역시간이 길다는 점이다[3,4]. 효과적인 스트릭트 니스 분석기는 고계함수, 복합 함수, 구조화 자료, 재귀함수에 대한 분석 시간이 충분히 빨라야 한다. 그러나 실험적인 연구 결과와 구현 경험들에 의하면 스트릭트 니스 분석으로 얻은 정보는 전체적인 수행 향상에 도움이 되는 경우도 있지만 고계 스트릭트 니스 정보는 거의 도움이 되지 않고, 복잡한 자료 구조에 대해서는 효과적이지 못하며, 번역시간도 스트릭트 니스 분석을 갖지 않는 번역 시간보다 거의 두배가 더 걸린다[4]. 따라서 스트릭트 니스 분석과 같은 번역시간 병렬성 표현방법과 함께 프로그래머에 의한 병렬성 표현 방법 즉 annotation도 연구되어 왔다.

프로그래머의 annotation에 의한 병렬성 표현 방법은 annotation을 신중하게 추가하지 않으면 계산의 의미가 변경되어 프로그램의 비종료을 초래할 수도 있다. 그러나 어느 프로그램에서나 병렬성의 주요한 자원은 프로그래머에 의해 유도된 알고리즘적 병렬성이므로 프로그래머 자신이 프로그램에 스트릭트 니스 정보를 바로 표기하는 것도 바람직한 방법이다. 한편 현존하는 함수 언어는 문제 정의에 관한 "what"은 매우 잘 표현하지만 실행시의 프로그램의 동작적 특성에 관한 "how" 부분이 결여되어 있는데, 프로그래머가 annotation을 사용하여 이 부분을 제어한다면 병렬 처리에 효율성을 기할 수도 있다.

병렬성 표현을 위한 annotation은 여러 연구[5-16]에서 제안되었는데, 계산 순서를 제어하기 위한 스케쥴링에 관련된 annotation과 특정한 프로세서에 태스크를 할당하기 위한 프로세서 사상에 관련된 annotation으로 크게 분류할 수 있으며, 스케쥴링 관련 annotation은 스트릭트 annotation과 프로세스 annotation으로 구분하기도 한다. 본 연구에서의 annotation은 스케쥴링 annotation에 해당하며 함수의 가인수와 함수 본체의 식에 annotation을 기술하여 프로그램 수행시 계산순서를 제어하게 된다.

Ⅲ. 원시 함수 언어의 정의

본 논문에서 정의한 원시 함수 언어는 함수 언어 Miranda[17]의 구문과 어의를 토대로 설계한 언어이다. 원어의 구문과의 가장 큰 차이점은 원어에 annotation 구문이 추가된 것이며, 원어의 구문 중에서 Guarded equation, 블럭구조(block structure), 고계함수(higher order function)에 대한 구문을 지원하며, 패턴매칭(pattern matching), ZF 표현(ZF expressions), 자료형 등에 대한 구문은 지원하지 않는다. annotation 구문 외에 추가된 구문은 블럭구조의 시작과 끝에 "{" 과 "}"을 추가한 것인데 이것은 구문 분석기 구현이 용이하고 프로그래밍시 분명하게 블럭을 표현하여 프로그램 독해성을 높일 수 있는 장점이 있기 때문이다.

annotation을 위한 구문은 함수의 가인수와 함수본체 내의 함수나 연산자에 표기하는 기호 "%"이다. 예를 들어 "f %a %b = (g (a b)) %+ (%h (a*2 b*2))"은 가인수 a와 b가 스트릭트하고, 함수 본체의 + 연산자의 두개의 인수 (g (a b))와 (h (a*2 b*2))이 스트릭트하며, 함수 h의 인수 (a*2)와 (b*2)도 역시 스트릭트함을 표현한 것이다.

원시 함수 언어의 구문을 EBNF로 표현하면 <표 1>과 같다.

Ⅳ. 번역기의 설계 및 구현

본 연구에서 설계하고 구현한 번역기는 함수 언어 번역기의 전반부(front-end)이다. 번역기는 그림 1과 같이 어휘 분석기, 구문 분석기, 확장 람다 계산 그래프 생성기(enriched lambda calculus graph generator)로 구성되며, 원시 프로그램을 구문 트리(AST; abstract syntax tree)인 확장 람다 계산 그래프로 번역한다. 번역 결과인 확장 람다 계산 그래프는 디버깅 파일로도

표 1 원시함수 언어의 구문

Table 1 The syntax of the source functional language

```
<SubMiranda> ::= '{' <equations> '}' <expr>
<equations> ::= <equation> | <equations> <equation>
<equation> ::= <id> <parameters> <equation body> <where clause>
<equation body> ::= '=' <expr> <guard>
<guard> ::= ε | ',' <boolean expr> <equation body>
<where clause> ::= ε | 'where' '{' <equations> '}'
<parameters> ::= ε | <parameters> <parameter>
<parameter> ::= <strict annotation> <id> |'(' <strict annotation> <id> <tuples> ')'
<tuples> ::= <tuple> | <tuples> <tuple>
<tuple> ::= ':' <id>
<expr> ::= <expr> <strict annotation> <arithmetic op> <expr>
         | <expr> <strict annotation> <list op> <expr> | '#' <expr>
         | <strict annotation> <id> <function call argument>
         | <constant> | '(' <expr> ')' | <list>
<list> ::= '[' <list element> ']' | 'null' | '"' <id> '"'
<list element> ::= <expr> | <list element> '..' <list element>
                 | <list element> ',' <list element>
<funtion call argument> ::= ε | '(' <argument list> ')'
<argument list> ::= <expr> | <argument list> <expr>
<boolean expr> ::= <boolean expr> <strict annotation> <logical op> <boolean exp>
                 | '~' <boolean expr> | '(' <boolean expr> ')' | <boolean term>
<boolean term> ::= <expr> <strict annotation> <relational op> <expr>
<arithmetic op> ::= '+' | '-' | '*' | '/'
<list op> ::= '++' | '--' | ':' | '!'
<logical op> ::= 'and' | 'or'
<relational op> ::= '==' | '~=' | '>' | '<' | '>=' | '<='
<id> ::= <letter> { <letter> | <digit> }*
<constant> ::= { <digit> }*
<letter> ::= [ 'a' - 'z' ] | [ 'A' - 'Z' ] | '_'
<digit> ::= ['0' - '9']
<strict annotation> ::= ε | '%'
```

출력되어 번역기 수행의 정확성을 분석하고 원시 프로그램의 오류를 찾아낼 때 사용된다.

어휘 분석기(scanner)는 Lex[18]를 이용하여 구현하며, 구문 분석기(parser)는 YACC[19]을 이용하여 구현하며, 확장 람다계산 생성기(enriched lambda calculus generator)는 C로 구현한다.

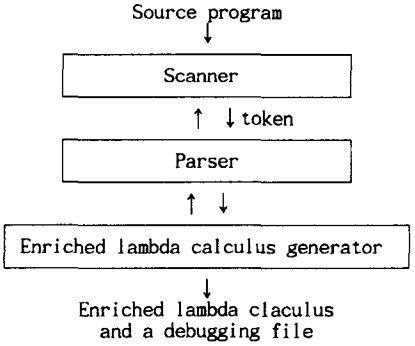


그림 1 번역기의 구성
Fig. 1 Structure of the translator

구문 분석기는 원시 프로그램을 구문 분석하여 문법적 오류를 점검하고 각 함수 정의식 마다 하나의 AST를 구성하여 AST 테이블에 등재한다. AST는 블럭구조 즉 지역 함수 정의문들을 표현하기 위하여 let-in 구조를 추가한 람다식이며 트리로 표현한다.

TAG	TokenValue	Pointer to left tree	Pointer to right tree

그림 3 AST의 노드 구조
Fig. 3 The abstract node structure of the AST

AST의 일반적인 형태는 그림 2와 같으며 원시언어의 구문이 중첩 블럭구조를 지원하므로 let-in 구조는 한 개의 AST내에 여러 개 나타날 수 있다.

AST의 각 노드는 그림 3과 같이 네 개의 항목으로 구성된다. 첫째 항목 TAG는 그 노드의 의미를 나타내며, 프로그램 상에 표기된 병렬성 정보도 또한 TAG로서 표현된다. 즉 가인수, 연산자, 함수에 대한 TAG는 각각 두개의 버전이 있다.

두 번째 항목 TokenValue에는 TAG에 따른 자료를 가지고 있는데 TAG가 정수를 나타내면 정수값, atom을 나타내면 문자열을 가지고 있으며 그 외에는 디버깅을 위한 문자열을 가진다. 세 번째와 네 번째 항목은 노드의 오른쪽과 왼쪽 노드를 가르키는 포

인터 항목이다.

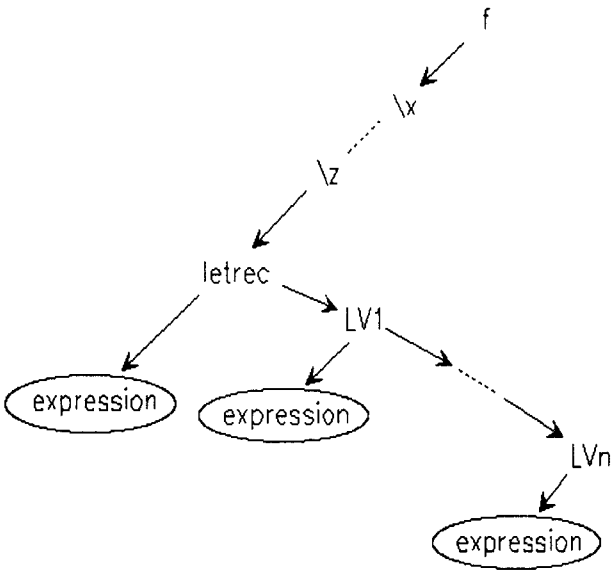


그림 2 AST의 형태
Fig. 2 The structure of the AST

V. 실험 및 분석

번역기는 UNIX 환경에서 구현하였으며, 번역기의 원시 프로그램의 구성과 각 원시 프로그램의 행의 길이는 표 2와 같다.

번역기의 정확성을 검토하기 위해서 다수의 시험용 프로그램을 작성하여 번역 결과를 분석하여 검토하였다.

표 2 번역기 원시프로그램의 구성
Table 2 Source programs of the translator

Module name	Number of lines	Comments
parer.y	400	YACC input file
scanner.l	139	Lex input file
parser.c	1177	YACC output file
scanner.c	905	Lex output file
lambda.c	1240	Enriched λ-calculus graph generator

번역 예로서 그림 4의 프로그램에 대한 번역 결과를 보이면 그림 5와 같다. 그림 5는 번역 결과인 확장 람다 계산 그래프를 전위 순회한 것을 텍스트로 표현한 것으로서, 디버깅 화일의 내용이다.

```
{   f %a %b = (g (a b)) %+ (%h (a*2
b*2))
}  f (1 2)
```

그림 4 예 프로그램
Fig. 4 An Example Program

Ⅵ. 결 론

함수 언어의 병렬성 표현을 위한 기법으로는 스트릭트니스 분석과 annotation이 있다.

본 논문은 annotation 기법에 관한 연구로서, annotation 구문을 갖는 원시 함수 언어를 설계하고 그것의 번역기 전반부를 구현하였다.

번역기는 UNIX 환경하에서 컴파일러 자동화 도구인 Lex와 YACC를 이용하여 구현하였다. 번역기는 원시 프로그램을 입력으로 받아서 번역한 후 중간 코드인 확장 람다 계산 그래프를 출력한다. 번역기의 정확성을 검토하기 위해서 다수의 시험용 프로그램을 작성하여 번역하고 그 결과를 분석 검토하여서 정확성을 확인하였다.

향후 과제는 번역기 후반부 구현에 대한 연구로서 현재의 구현을 토대로 번역기의 후반부를 구현하고 있다[21].

```
-- file name : sample4.deb

#Welcome to the SubMiranda compiler !

#Pass 1 -- Parsing & making lambda-calculus graph

ASTtable[ 0 ] = function( f ) :

(100 f) l (165 a) l (165 b) l (50 @) l (50 @) l (204 ADD) l (0)
r (0)
r (50 @) l (50 @) l (59 g) l (0)
r (0)
r (59 a) l (0)
r (0)
r (59 b) l (0)
r (0)
r (50 @) l (50 @) l (202 h) l (0)
r (0)
r (50 @) l (50 @) l (6 MULT) l (0)
r (0)
r (59 a) l (0)
r (0)
r (0 2) l (0)
r (0)
r (50 @) l (50 @) l (6 MULT) l (0)
r (0)
r (59 b) l (0)
r (0)
r (0 2) l (0)
r (0)
r (0)
r (0)
r (0)

ASTtable[ 1 ] = function( @ ) :

(50 @) l (50 @) l (59 f) l (0)
r (0)
r (0 1) l (0)
r (0)
r (0 2) l (0)
r (0)

#Compilation finished !
```

그림 5 예 프로그램의 번역 결과
Fig. 5 The result of translating the example program

참 고 문 헌

[1] P. Hudak, "Conception, evolution and application of functional programming languages", ACM Computing Survey, Vol.21, No.3, pp.359-411, 1989

[2] J. Backus, "Can programming be liberated from the von neumann style? A functional style and its algebra of programs", CACM, Vol. 21, No. 8, pp.613-641, 1978

[3] S. L. Peyton Jones, "Parallel implementation of functional programming language", The computer journal, Vol.32, No.2, pp. 175-186, 1989

[4] K. Hammond, "Parallel Functional Programming: An Introduction", PASCO' 94, PP.1-13, 1994

[5] P. Hudak, "Denotational Semantics of a Para Functional Programming Language", International Journal of Parallel Programming, Vol. 15, No. 2, 103-125, 1986

[6] P. Hudak, "Para Functional Programming", IEEE COMPUTER, pp. 60-70, 1986

[7] A. Bloss, P. Hudak, Y. Young, "An optimizing compiler for a modern functional language", The computer journal, Vol.31, No.6, pp.152-161, 1988

[8] B. k. Szymanski, 'Parallel Functional Languages and Compilers', ACM press, 1991

[9] F. W. Burton, "Annotation to Control Parallelism and Reduction Order in the Distributed Evaluation of Functional Programs", ACM Trans. on Prog. Lang. and Sys., Vol.6, No.2, April, pp.159-174, 1984

[10] P. Henderson, 'Functional Programming Application and Implementation', Prentice Hall, 1980

[11] J. Schwarz, "Using Annotation to Make Recursion Equations Behave", IEEE Trans. on Soft. Eng., Vol. SE-8, No.1, Jan., pp.21-33, 1982

[12] R. H. Halstead, "Multilisp: A Language for Concurrent Symbolic Computation", ACM Trans. on. Prog. Lang. and Sys., Vol. 7, No. 4, Oct., pp.501-538, 1985

[13] L. Augustsson, T. Johnsson, "The chalmers lazy ML compiler", The computer journal, Vol. 32, No. 2, 1989

[14] Luc Maranget, "GAML : A Parallel Implementation of Lazy ML", Springer Verlag LNCS 523, pp. 102-123, 1991

[15] E.G.J.M.H. Nocker, J.E.W. Smetsers, M.C.J.D. Eekelen, M.J. Plasmeijer, "CONCURRENT CLEAN", Springer Verlag LNCS 506, pp. 202-219, 1991

[16] P. Roe, "Some ideas on parallel funcional programming", Proc. of the 1989 Glasgow Workshop, pp. 338-352, 1989

[17] D. A. Turner, "Miranda: A non-strict functional language with polymorphic types", Springer-Verlag LNCS 201, pp. 1-17, 1985

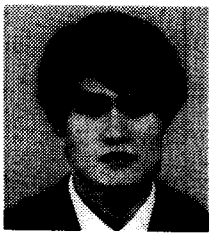
[18] M. E. Lesk, "Lex - A lexical analyzer generator", Computing science techinical report #39, Bell Lab., 1975

[19] S. C. Johnson, "Yacc - Yet Another Compiler - Compiler", Computing science techinical report #39, Bell Lab., 1975

[20] S. L. Peyton Jones, 'The implementation of the functional programming language', Prentice Hall International, 1987

[21] 이종희, 최관덕, 윤영우, 강병욱, "지연 함수 언어 Miranda의 G -기계 기반 번역기 개발", 한국 정보 처리 학회 정보 처리 논문지, 제2권 제5호, pp. 733-745, 1995. 9

□ 筆者紹介



최관덕

1989년 영남대학교 전산공학과(공학사)

1991년 영남대학교 전산공학과(공학석사)

1995년 영남대학교 전산공학과 박사과정 수료

1996년 ~ 현재 대구전문대학 전자계산과 전임강사