지연함수언어 Miranda의 G-기계 기반 번역기 개발

이 종 희[†] 최 관 덕[††] 윤 영 우[†††] 강 병 욱[††††]

요 약

본 연구는 함수언어의 번역기 개발을 목적으로 한다. 이를 위하여 지연어의를 갖는 원시함수 언어를 정의하고 그것의 번역기를 설계, 구현, 평가한다. 함수프로그램의 실행모형은 G-기계를 기반으로 한 컴비네이터 그래프축소이다. 번역기는 전체 4단계로 구성되며 원시프로그램을 C를 사용한 목적프로그램으로 번역한다. 번역기의 첫 번째 단계에서는 원시프로그램을 확장람다계산 그래프로 번역하고, 두 번째 단계에서 수퍼컴비네이터그래프로 변환하고, 세 번째 단계에서 G-기계어 프로그램으로 번역하고, 마지막 단계에서 G-기계어 프로그램을 C로 번역한다. 생성된 목적 프로그램은 C 컴파일러에 의해서 실행 프로그램으로 번역된다. 번역기 구현은 UNIX환경에서 컴파일러 자동화 도구인 YACC, Lex를 이용하여 구문분석기, 어휘분석기를 구현하고, 그 외의 루틴은 C로 구현한다. 본 논문에서는 번역기에 사용된 구현기법과 수행결과를 기술한다.

Development of a G-machine Based Translator for a Lazy Functional Programming Language Miranda

Jong-Hee Lee,[†] Kwan-Deok Choi,[††]
Young-Woo Yoon,[†††] Byong-Ug Kang[††††]

ABSTRACT

This study is aimed at construction of a translator for a functional programming language. For this goal we define a functional programming language which has lazy semantics and develop a translator for it. The execution model selected is the G-machine-based combinator graph reduction. The translator is composed of 4 phases and translates a source program to a C program. The first phase of the translator translates a source program to a enriched lambda-calculus graph, the second phase transforms a lambda-calculus graph into supercombinators, the third phase translates supercombinators to a G program and the last phase translates the G program to a C program. The final result of the translator, a C program, is compiled to an executable program by C compiler. The translator is implemented in C using compiler development tools such as YACC and Lex, under the UNIX environments. In this paper we present the design and implementation techniques for developing the translator and show results by executing some test problems.

1. 서 론

함수언어는 단순한 상태전이규칙(state transition rule)을 가지는 적용계산모형(applicative computing model)의 언어로서 표현력이 풍부하고, 어의(semantics)가 단순하여 프로그램의 정확성 증명이 쉬우며, Church-Rosser 성질을 가지므로 병렬처리에도 적합하다는 장점을 가진다. 그러나 이런 장점에도 불구하고 함수언어는 현재 널리 사용되고 있지는 않은데, 그 한 가지 이유

† 정 회 원:영남전문대학 전자계산기과 교수
†† 정 회 원:영남대학교 대학원 전산공학과 컴퓨터시스템전공(박사과정수료)
††† 정 회 원:영남대학교 공과대학 전산공학과 부교수
†††† 정 회 원:영남대학교 공과대학 전산공학과 교수
논문접수:1995년 4월 3일, 심사완료:1995년 9월21일

는 언어의 개념이 표현력과 어의의 단순성에 초점이 맞추어져 있는 반면에, 현재의 폰노이만형 컴퓨터의 동작특성을 반영하지 못하기 때문에 효율적인 번역기 구현이 어려워 실행속도가 느리다는 단점이 있기 때문이다. 이런 단점을 해결하기 위한 연구로서 새로운 비-폰노이만형 컴퓨터를 개발하여 실행속도를 개선하기 위한 하드웨어적인 연구와 함수언어를 기존의 폰노이만형 컴퓨터에서 효과적으로 실행하기 위한 소프트웨어적인 연구가 있다[1, 2].

소프트웨어적인 연구로는 현재 환경기반방식(environment-based scheme)과 그래프축소(graph reduction)라는 구현방법이 매우 효과적이라고 알려져있다[3, 4]. 이 두 가지 구현방법은 함수 적용시 가인수에 실인수를 전달하는 방법에 따라 구분되는데, 환경기반방식은 환경을 통해서 전달하고, 그래프축소는 프로그램 그래프의 해당 노드를 포인트하여 전달한다. 특히 그래프축소는 프로그램을 그래프로 표현하고 그래프를 변환함으로써 프로그램을 실행하는 방법으로 부분식의 공유와 지연실행(lazy implementation)이 쉽다. 그래프축소는 다시 람다계산(lambda calculus)을 기반으로 한 구현과 컴비네이터(combinator)를 기반으로 한 구현으로 나눌 수 있다. 컴비네이터 구현은 람다식(lambda expression)을 자유변수를 포함하지 않는 식으로 변환하는 구현기법으로서 자유변수가 없으므로 그래프축소를 기계어 수준으로 구현하는데 적합하다. 그래프 축소를 기계어 수준으로 구현하기 위한 연구는 추상기계(abstract machine)의 설계에 바탕을 둔다. 추상기계는 각 추상기계어(abstract machine code)에 대한 상태전이규칙과 함수프로그램을 일련의 추상기계어들로 번역하는 번역규칙으로 설계된다[3].

본 연구는 함수언어의 소프트웨어적인 연구로서, 함수언어의 기계어 수준 구현을 목적으로 한다. 이를 위하여 지연어의(lazy semantics)를 갖는 원시함수언어를 정의하고 그것의 번역기를 개발한다. 함수프로그램의 실행모형은 G-기계[5, 6, 7, 8]를 기반으로 한 컴비네이터 그래프축소이다. 번역기는 4단계로 구성하며, 각 단계의 번역결과인 중간언어는 확장람다계산(enriched lambda calculus), 수퍼컴비네이터(supercombinator), G-기계어(G-code)이며, 번역기의 목적어는 C이다.

본 논문의 2장에서는 그래프축소를 기반으로 하는 함수언어의 번역기법에 대해서 기술하고, 3장에서는 정의한 원시함수언어의 구문 및 특징을 보이며, 4장에서는 번역기의 각 단계에 사용된 구현기법을 설명하고, 5장에서는 번역기에 대한 실험 및 분석으로서 몇가지 시험프로그램에 대한 번역의 정확성과 수행속도를 보이며, 6장에서는 결론 및 향후과제를 논한다.

2. 그래프축소를 기반으로 하는 함수프로그램의 번역기법

그래프축소는 1971년 Wadsworth에 의해 제안된 것으로서 프로그램을 그래프로 표현하고 그래프를 변환하는 과정으로 프로그램을 실행하는 계산모형이다. 그래프축소는 번역된 그래프의 형태에 따라 람다계산을 기반으로 하는 그래프축소와 컴비네이터를 기반으로 하는 그래프축소로 나눌 수 있다. 람다계산은 간단한 구문과 단순한 어의를 가지는 계산모형으로서 모든 함수 프로그램을 표현할 수 있기 때문에 함수언어 자체나 번역시의 중간언어로서 사용된다. 람다계산을 기반으로 하는 그래프축소에서 사용되는 축소규칙에는 함수의 가인수에 실인수를 치환하는 β-축소규칙과 기본함수를 실행하는 σ-축소규칙 및 변수명 충돌을 해결하기 위한 α-축소 등이 있다[3].

람다계산을 기반으로 하는 그래프축소를 좀 더 효율적으로 구현하기 위해서는 개개의 람다본체(lambda-body)를 고정된 명령어의 나열 즉 기계어 프로그램으로 번역해야하는데, 람다식내에 자유변수가 존재하면 자유변수의 값을 얻는 메커니즘을 추가해야하므로 구현에 어려움이 있다. 이를 해결한 구현이 컴비네이터를 기반으로 하는 구현이다. 컴비네이터란 자유변수가 하나도 없는 람다식을 말하며, 고정컴비네이터(fixed-set combinator), 프로그램유도컴비네이터(program-derived cominator) 등이 있다[3].

고정컴비네이터 번역기법은 원래의 람다식을

자유변수가 없는 람다식으로 변환할 때에 고정된 종류의 컴비네이터만을 사용하는 기법을 말한다. 기본적인 컴비네이터는 S, K, I등이며, 여기에 몇몇 최적화 컴비네이터를 추가한 Turner의 컴비네이터[9]가 고정컴비네이터 번역기법의 대표적인 기법이다. Turner 컴비네이터 번역기법은 변환된 그래프가 원래 그래프의 모습과 너무 달라서 디버깅이 어렵고 실행단계가 매우 작은 단계로 쪼개져서 결합의 부담이 큰 단점을 가지고 있으나, 완전지연평가(fully lazy evaluation) 기능이 있다[9, 10].

프로그램유도컴비네이터 번역기법은 일반적으로 람다식을 람다승급(lambda-lifting)[6,10]하여 컴비네이터로 변환하는 번역기법이다. 람다승급시 자유변수만을 승급하는 번역기법에서는 완전지연평가를 구현할 수는 없으나, Hughes의 수퍼컴비네이터 번역기법[10]에서는 람다승급시 자유변수만을 승급하지 않고 최대자유식(mfe: maximally free expression)을 승급하므로써 완전지연평가 기능을 제공한다.

그래프축소의 기계어 수준 구현에 관한 연구는 일반적으로 추상기계의 설계에 바탕을 둔다.

추상기계는 각 추상기계어에 대한 상태전이규칙과 함수프로그램을 일련의 추상기계어들로 번역하는 번역규칙으로 설계된다[3]. 추상기계를 기반으로 하는 번역기 구현에서는 원시프로그램을 추상기계어로 먼저 번역하며 이를 다시 목적기계의 기계어로 번역한다. 현재까지 연구된 추상기계로는 G-기계(G-machine)[5, 6, 7, 8], TIM[11], Spineless G-machine[12], Categorical Abstract Machine[13], Spineless Tagless G-machine[14] 등이 있다.

본 연구에서 사용하는 G-기계는 스웨덴의 찰머스공과대학의 Johnsson과 Augustsson에 의해 개발된 추상기계이다. G-기계는 5개의 요소로 구성된 상태로 표현되는 유한상태기계로서, 5개의 요소는 출력장치로의 출력을 나타내는 O, 현재실행중인 코드를 나타내는 C, 힙(heap)에 구성되는 그래프 G, 스파인 스택(spine stack) S, 그래프축소중 문맥절환시 현재의 문맥을 저장하기 위한(S,C)쌍으로 구성된 덤프 스택(dump stack) D이다. G-기계를 기반으로 한 번역기법은 컴비네이터로 구성된 프로그램을 일련의 G-기계어로 번역하는 기법으로서 번역된 기계어 프로그램은 지연그래프축소 사이클 즉, unwind-instantiate-update을 순서대로 기술한 일련의 추상기계어 표현이다. 여기서 unwind는 평가할 함수를 찾기 위해서 그래프의 적용노드(apply node)를 왼쪽으로 타고 내려가면서 각 적용노드를 스파인 스택에 보관하는 과정이며, instantiate는 unwind 과정에서 찾은 함수의 가인수에 스파인 스택에 보관되어 있는 실인수에 대한 포인터를 대치하고, redex(reducible expression)를 WHNF(Weak Head Normal Form)으로 평가하는 과정이며, update는 평가결과를 원래식의 루트노드에 덧쓰는 과정이다[5, 6, 7, 8].

3. 함수언어 정의

본 논문에서 정의하여 구현할 함수언어는 함수언어 Miranda™[15]의 구문과 어의를 토대로 설계한 Miranda subset 언어이다. Miranda의 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. 지원하는 구문 중에서 원어의 구문과 다른 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. 설계한 함수언어의 구문을 EBNF로 표현하면 〈표 1〉과 같으며, 이 언어를 사용한 프로그램의 예는 부록 3과 같다.

4. 번역기의 구현

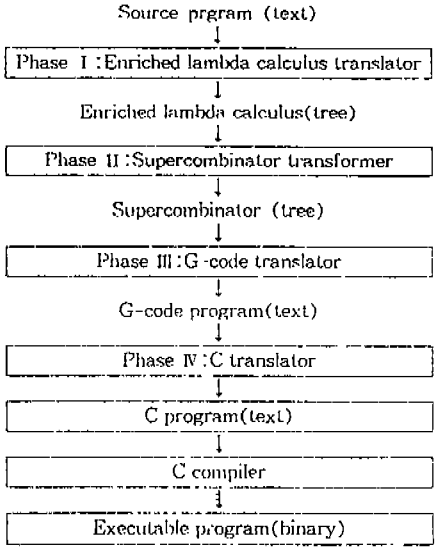
번역기는 (그림 1)과 같이 전체 4단계로 구성된다. 첫 번째 단계에서는 원시프로그램을 구문분석하여 구문오류를 점검하고 구문트리(AST ; abstract syntax tree)인 확장람다계산그래프(enriched lambda calculus graph)로 번역한다. 두 번째 단계에서는 확장람다계산그래프를 수퍼컴비네이터그래프(supercombinator graph)로 번

환한다. 세 번째 단계에서는 수퍼컴비네이터그래프를 G-기계어 프로그램으로 번역하며, 네 번째 단계에서 G-기계어 프로그램을 C로 번역한다. 번역결과인 C 프로그램은 기존의 C 컴파일러에 의해서 실행프로그램으로 번역된다.

〈표 1〉 원시함수언어의 구문

〈Table 1〉 The syntax of the source functional programming 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> ::=<parameter> | <parameters>
                <parameter>
<parameter> ::=ε | <id> | '(' <id> <tuples> ')'
<tuples> ::=<tuple> | <tuples> <tuple>
<tuple> ::=',' <id>
<expr> ::=<expr> <arithmetic op> <expr> | <expr>
          <list op> <expr> | '#'<expr>
          | <id> <function call argument> | <constant>
          '(' <expr> ')' | <list>
<list> ::= '[' <list element> ']' | 'null' | '"'<id> '"'
<list element> ::=<expr> | <list element>','<list element>
                | <list element> ';' <list element>
<function call argument> ::=ε | '('<argument list> ')'
<argument list> ::=<expr> | <argument list> <expr>
<boolean expr> ::=<boolean expr> <logical op>
                <boolean exp> | '~'<boolean expr> |
                '('<boolean expr>')' | <boolean term>
<boolean term> ::=<expr> <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']
```

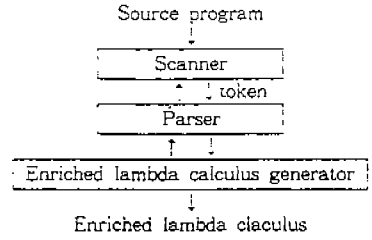


(그림 1) 번역기의 구성

(Fig. 1) Overall structure of the translator

4.1 단계 1: 확장람다계산그래프 번역기

원시프로그램을 구문분석하고 확장람다계산그래프로 번역하는 단계로서 (그림 2)와 같이 구성된다. 어휘분석기(scanner)는 Lex[16]를 이용하여 구현한다. 구문분석기(parser)와 확장람다계산 생성기(enriched lambda calculus generator)는 YACC[17]을 이용하여 구현하며 문법지시적변환(syntax-directed translation) 기법으로 원시프로그램을 확장람다계산그래프로 번역한다. 구문분석기는 원시프로그램을 구문분석하여 문법적 오류를 점검하고 각 함수정의식 마다 하나의 AST를 구성하여 AST테이블에 등재한다. AST는 블럭구조 즉 지역함수정의문들을 표현하기 위하여 let-in 구조를 추가한 람다식이며 트리로 표현한다.



(그림 2) 단계 1의 구성

(Fig. 2) The configuration of the phase I

AST의 일반적인 형태는 (그림 3)과 같으며 원시언어의 구문이 중첩 블럭구조를 지원하므로 let-in 구조는 한개의 AST내에 여러개 나타날 수 있다. AST의 각 노드는 (그림 4)와 같이 네 개의 항목으로 구성되는데, 첫째 항목 TAG는 그 노드의 의미를 나타내며, 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 포인터 항목이다.

예로서 (그림 5)의 프로그램은 (그림 6)과 같이 번역된다. (그림 6)에서 볼 수 있듯이, 이 단계에서 지역변수는 각기 유일한 이름으로 재명명되는데 이 작업으로서 다음 단계의 알고리즘 수행시의 변수명 충돌문제(name clash problem)[3]를 자동적으로 해결한다.

4.2 단계 2 : 수퍼컴비네이터 변환기

확장람다계산그래프를 수퍼컴비네이터그래프로 변환하는 단계이며 (그림 7)과 같이 3 패스로 구성된다. 첫 번째 패스인 Super는 확장람다계산그래프를 수퍼컴비네이터로 변환하며, 두 번째 패스인 Flattener는 수퍼컴비네이터의 모든 지역변수들을 최상위로 이동시키는 패스이며, 세 번째 패스인 Optimizer에서는 수퍼컴비네이터를 최적화한다.

수퍼컴비네이터 변환 알고리즘은 Hughes의 수퍼컴비네이터 번역기법[3,10]에 기반을 둔다. 수퍼컴비네이터 번역기법은 람다식을 컴비네이터로 변환할 때에 자유변수만을 승급하지 않고 최대자유식을 승급하므로써 완전지연평가를 제공하며, 컴비네이터의 크기가 커서 그래프축소시 축소 회수가 적은 이점이 있다.

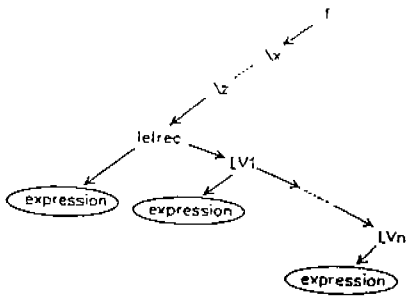
수퍼컴비네이터 변환 알고리즘은 다음과 같다.

모든 확장람다계산그래프 즉 AST에 대하여 다음 ①~⑧을 수행한다.

① 하나의 AST를 선택한다.

F=…E…

② F내에 지역정의식들이 존재하면 지역정의식들 각각에 대해서 ②~⑦를 수행한다. 수행결과로 지역정의식은 지역변수식들이 된다.



(그림 3) AST의 형태
(Fig. 3) The structure of the AST

TAG	TokenValue	Pointer to left tree	Pointer to right tree

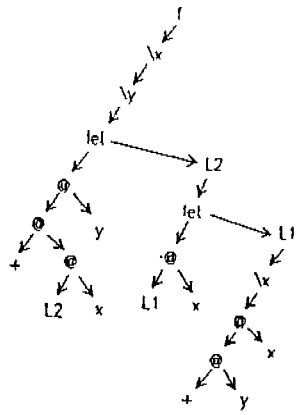
(그림 4) AST의 노드 구조
(Fig. 4) The abstract node structure of the AST

```
{f x y = g(x) + y
       where { g = h(x)
                   where { h x = y + x }}
}f (1 1)
```

(그림 5) 예 프로그램
(Fig. 5) An example program

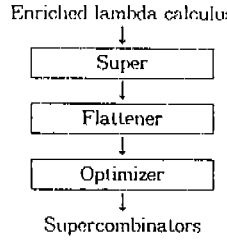
```
f = λx. λy. let l2 = let l1 = λx. + y x
                     in l1 x
            in + (l2 x) y
result = f 1 1
```

(a) the textual representation



(b) the graphical representation

(그림 6) 예 프로그램의 AST 형태
(Fig. 6) The AST for the example program



(그림 7) 단계 2의 구성
(Fig. 7) The configuration of the phase II

$$F = \cdots \text{let } LV_1 = \cdots$$
$$\cdots$$
$$LV_n = \cdots$$
$$\text{in } E$$

지역정의식내의 모든 람다식에 대하여 ③~⑦을 수행한다.

③ 가장 안쪽의 람다식을 선택한다. 람다식이 없으면 ⑧을 수행한다.

$$L = \lambda v.\ exp$$

④ 최대자유식을 찾는다. 최대자유식은 속박변수를 포함하지 않는 부분식(subexpression)이며, 속박변수는 현재의 람다변수와 현재의 람다식내의 지역변수들이다.

$$mfe = \{e_1, \cdots, e_n\}$$

⑤ 최대자유식의 순서를 식에 포함된 자유변수의 어휘적 순서에 따라 나열한다. 이런 순서매김의 결과로 최적화 작업의 효율을 높일 수 있다.

$$mfe = \{e_1, \cdots, e_n\}$$

⑥ 새로운 컴비네이터를 정의한다. 아래 식에서 [x/y]은 y를 x로 대치한다는 의미이다.

$$\$\,newcomb\ i_1 \cdots i_n$$
$$v = exp[i_1/e_1, \cdots, i_n/e_n]$$

⑦ 현재의 람다식을 새로운 컴비네이터에 mfe를 적용한 식으로 대치한다.

$$L = \$\,newcomb\ e_1, \cdots, e_n$$

⑧ 현재의 함수정의식이 지역함수정의식이 아니면 남아있는 식을 컴비네이터로 만든다.

$$\$F = \cdots$$

(그림 5)의 프로그램의 수퍼컴비네이터 변환 결과는 (그림 8)과 같다. 그림에서 p_n은 n번째가인수, l_n은 n번째 지역변수, C_x는 수퍼컴비네이터 x를 의미한다.

(그림 8)의 수퍼컴비네이터 정의들은 Flatten-

```
C_0 p1 p2 = +P1 P2
C_1 p1 p2 = let l2 = let l1 = C_0 p2 in l1 p1
            in + (l2 p1) p2
C_2 p1 = C_1 p1
C_f = C_2
C_4 = C_f 1 1
C_Prog = C_4
```

(그림 8) 예 프로그램에 대한 수퍼컴비네이터 정의

(Fig. 8) Supercombinators for the example program

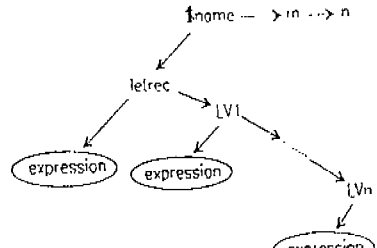
er 패스에 의해서 (그림 9)와 같이 변환된다. 모든 지역변수들을 최상위로 변환시키는 이유는 다음 단계인 G-코드 번역에 용이하기 때문이며, 이 과정에서 지역변수들은 재명명된다.

```
C_0 p1 p2 = +p1 p2
C_1 p1 p2 = let lv1 = lv2 p1
                lv2 = C_0 p2
            in+ (lv1 p1) p2
C_2 p1 = C_1 p1
C_f = C_2
C_4 = C_f 1 1
C_Prog = C_4
```

(그림 9) 예 프로그램의 flattening 후의 수퍼컴비네이터

(Fig. 9) Supercombinators after flattening for the example program

Optimizer 패스의 최적화 내용은 동일한 형태의 컴비네이터를 제거하고 상수 정의식을 제거하는 것이다. 이 패스에서는 또한 다음 패스의 알고리즘 수행을 용이하게 하기 위해서 수퍼컴비네이터 정의에 관한 정보를 수퍼컴비네이터그래프에 추가한다. 그 정보는 가인수의 개수와 지역변수정의의 개수이며, 이런 정보를 추가한 수퍼컴비네이터그래프의 형태는 (그림 10)과 같다. (그림 10)에서 m은 가인수의 개수를 n은 지역변수정의의 개수를 의미한다. (그림 9)의 최적화 결과는 (그림 11)과 같다.



(그림 10) 수퍼컴비네이터그래프의 구조

(Fig. 10) The structure of the supercombinator graph

```
C_0 2 2 = let lv1 = lv2 p1
              lv2 = + p2
          in+ (lv1 p1) p2
C_1 0 0 = C_0 1 1
C_Prog 0 0 = C_1
```

(그림 11) 최적화 후의 수퍼컴비네이터

(Fig. 11) Supercombinators after optimizing

4.3 단계 3 : G-기계어 프로그램 번역기

수퍼컴비네이터그래프를 G-기계어로 번역하는 단계이다. 목적기계인 G-기계는 유한상태기계로서 상태는 5개의 요소로 구성된다. 5개의 요소는 출력장치로의 출력을 나타내는 O, 현재실행중인 코드를 나타내는 C, 힙에 구성되는 그래프 G, 스파인 스택 S, 그래프축소중 문맥절환시 현재의 문맥을 저장하기 위한 (S,C)쌍으로 구성된 덤프 스택 D이다. 따라서 G-기계의 한 상태는 5-튜플 ⟨O, C, S, G, D⟩로 표현한다. 각 G-기계어에 대한 상태전이규칙은 부록 1과 같으며, 수퍼컴비네이터그래프를 G-기계어로 번역하는 번역규칙은 부록 2와 같다.

이 단계의 결과인 G-기계어 프로그램은 전체적으로 3 부분으로 구성된다. 첫 번째 부분은 G-기계를 초기화하는 부분으로 G-기계어 BEGIN으로 표현되며, 두 번째 부분은 그래프축소를 시작하고 종료하는 부분으로 G-기계어열 PUSHGLOBAL C_Prog ; EVAL ; PRINT ; END이며, 세 번째 부분은 수퍼컴비네이터 각각에 대한 G-기계어 열이다. 내정컴비네이터에 대한 정의는 목적기계의 라이브러리 루틴으로 구성할 수 있으므로 G 프로그램에는 포함하지 않는다. 번역 예로 (그림 11)의 컴비네이터들을 G-기계어로 번역한 결과는 (그림 12)와 같다.

```
BEGIN;
PUSHGLOBAL C_Prog ; EVAL ; PRINT ;
END ;

GLOBSTART C_0 2 ;
ALLOC 2 ; PUSH 2 ; PUSH 1 ; MKAP 1 ; UPDATE2 ; PUSH 3 ;
PUSHGLOBAL ADD ; MKAP 1 ; UPDATE 1 ; PUSH 3 ;
  EVAL ; GET ;
PUSH 3 ; PUSH 3 ; MKAP 1 ; EVAL ; GET ; ADD ; UPDINT
  5 ; POP 4 ;
RETURN ;

GLOBSTART C_1 0 ;
PUSHINT 1 ; PUSHINT 1 ; PUSHGLOBAL C_0 ; MKAP 2 ;
UPDATE 1 ; POP 0 ;
UNWIND ;

GLOBSTART C_Prog 0 ;
PUSHGLOBAL C_1 ; EVAL ; UPDATE 1 ; POP 0 ;
UNWIND ;
```

(그림 12) 수퍼컴비네이터 C_0에 대한 G-기계어 프로그램

(Fig. 12) The G-code program for the supercombinator in (Fig. 11)

4.4 단계 4: C 프로그램 번역기

G-기계어 프로그램을 C로 번역하는 단계이다. 이를 위하여 G-기계의 5 요소를 다음과 같이 사상한다. 출력 O는 화면장치로, 코드 C는 C 언어의 문으로, 스택 S는 정적 메모리 내의 배열로, 그래프 G는 힙에 동적으로 구성되며, 덤프 D는 시스템 스택으로 사상한다. G-기계어 프로그램의 3 부분 중에서 첫 번째와 두 번째 부분은 main 함수로 번역되며, 세 번째 부분은 수퍼컴비네이터마다 하나의 C 함수로 구성되며, 각 부

```
int C_0( ), C_1( ), ..., C_Prog( ) ; /* 수퍼컴비네이터 함수에
대한 함수원형 */
int(*SuperComb[ ])( ) ={C_0, C_1, ..., C_Prog} ; /* 수퍼컴비
네이터 함수 호출을 위한 정의 */
int SuperArgs[3]={2, 0, ..., 0} ; /* 수퍼컴비네이터 함수의 가인수
개수 */
#include "g_deflib.h"/*라이브러리 루틴*/
main( )/*BEGIN*/
{
        begin_of_session();
        /*PUSHGLOBAL C_Prog*/
        sp++ ;
        stack[sp].right=make_t_cell(_comb) ;
        stack[sp].right->v.ival=2 ;
        /*EVAL*/
        switch(stack[sp].right->tag) {
          case_apply : /*unwind*/...
          case_comb : /*COMB 0 C*/...
            .
            .
            .
          default : /*built_in Comb*/
                  break;
        }
        /*PRINT*/
        built_in_comb_print( ) ;
        /*END*/
        end_of_session( ) ;
        return ;
}

C_0( )/*GLOBSTART C_0 2*/
{
        /*2(rearrange stack)*/
stack[sp-0].right=stack[sp-1].right->right ;
stack[sp-1].right=stack[sp-2].right->right ;
          .
          .
          .
}

C_1( ) {...}

        .
        .
        .

C_Prog( ) {...}
```

(그림 13) C 프로그램의 구조

(Fig. 13) The structure of the C program

분을 구성하는 G-기계어 열들은 해당 C 언어의 문들로 번역된다.

번역결과인 C 프로그램의 구성은 (그림 13)과 같이 개략적으로 나타낼 수 있는데, 프로그램의 시작부분은 수퍼컴비네이터 정의에 해당하는 함수를 호출하기 위한 정의문들과 라이브러리 루틴을 포함하는 문으로 구성되며, 계속해서 G-기계어 프로그램의 첫 번째 부분에 대한 G-기계어 열이 main 함수의 루틴으로 구성되며, 그 다음부터 각 수퍼컴비네이터에 대한 함수들이 정의된다.

번역결과인 C 프로그램은 헤더화일 g_deflib.h에 정의된 라이브러리 루틴과 함께 C 컴파일러를 사용하여 실행 프로그램으로 번역한다. 라이브러리 루틴에는 전역변수로 스택 포인터 sp와 임시 포인터 변수 tmp의 정의, 메모리 관리 루틴, eval, print, add, sub 등의 내정컴비네이터에 대한 함수들을 포함하며 그 구성은 (그림 14)와 같다.

이 단계의 번역 예로서 (그림 12)의 G-기계어열은 (그림 15)와 같은 C 함수로 번역된다. 그림에서… 으로 기술된 부분은 앞부분에 비슷한 문들이 보이기 때문에 지면관계상 생략한 부분이다. 그림에서 볼 수 있듯이 각 수퍼컴비네이터 정의에 대한 함수는 시작부분에 스택을 재 정렬하는 문들로 시작하여 각각의 G-기계어에 대한 루틴이 C 문들로 기술된다.

```
#include<stdio.h>
#include<string.h>
#define—false 0/*태그 정의*/
#define—true 1
#define—add 0
#define—div 1
 ...
/*내정컴비네이터에 대한 함수 정의 프로토타입*/
int add( ), div( ), mult( ), sub( ), and( ), not( ), or( ), eq( ),
    ge( ), gt( ), le( ),
lt( ), ne( ), cons( ), head( ), mm( ), pp( ), sharp( ), subs( ).
    tail( ), if—( ), k—2—1( ), k—2—2( );
/*내정컴비네이터 함수 호출를 위한 정의*/
int(*Comb[23])( )={add, div, mult, sub, and, not, or, eq, ge, gt,
    le, lt, ne, cons, head, mm, pp, sharp, subs, tail, if—, k—2—1,
    k—2—2};
/*셀 정의*/
typedef struct cell {
      unsigned char tag;
      union {int ival; char cval[8];} v;
      struct cell *left;
      struct cell *right;
} Cell, *CellPtr;
/*전역 변수 정의*/
Cell stack[stacksize];
Cell *tmp;
int  itmp;
/*메모리 관리 함수 정의*/
Cell *get—cell() {…}
Cell *make—a—cell(atom) char atom[8]; {…}
Cell *make—i—cell(ival) int ival; {…}
Cell *make—t—cell(tag) unsigned char tag; {…}
 ...
/*수행시간 라이브러리 함수 정의*/
begin—of—session( ){…}
end—of—session( ){…}
runtime—error(string) char *string; {…}
debug—routine( ){…}
 ...
/*내정 컴비네이터에 대한 함수 정의*/
built—in—comb—print( ){…}
print( ){…}
eval( ){…}
add( ){…}
 ...
K—2—2( ){…}
```

(그림 14) 라이브러리 루틴
(Fig. 14) Library routines

```
C—0() /*GLOBSTART C—0 2*/
{
        /*2(rearrange stack)*/
        stack[sp—0].right=stack[sp—1].right—>right;
        stack[sp—1].right=stack[sp—2].right—>right;

        /*ALLOC 2*/
        sp——;
        stack[sp].right=get—cell( );
        sp——;
        stack[sp].right=get—cell( );

        /*PUSH 2*/
        sp++;
        stack[sp].right=stack[sp—2—1].right;

        /*PUSH 1*/…

        /*MKAP 1*/
        tmp=make—t—cell(—apply);
        tmp—>left=stack[sp].right;
        tmp—>right=stack[sp—1].right;
        stack[sp—1].right=tmp;
        sp——;

        /*UPDATE 2*/
        stack[sp—2].right—>tag=stack[sp].right—>tag;
        switch(stack[sp].right—>tag) {
            case—int:
            case—comb: stack[sp—2].right—>v.ival=stack[sp].
                        right—>v.ival; break;
            default: strcpy(stack[sp—2].right—>v.cval,
                    stack[sp].right—>v.cval);
        }
        if(stack[sp].right—>right) stack[sp—2].right—>right=
        stack[sp].right—>right;
        if (stack[sp].right—>left) stack[sp—2].right—>left=
        stack[sp].right—>left;
        sp——;

        /*PUSH 3*/…

        /*PUSHGLOBAL ADD*/
        sp++;
        stack[sp].right=make—t—cell(—add);

        /*MKAP 1*/…

        /*UPDATE 1*/…

        /*PUSH 3*/…

        /*EVAL*/
        switch (stack[sp].right—>tag) {
```

```
case --apply : /*unwind*/
               while (1) {
                   sp++ ;
                   stack[sp].right=stack[sp-1].right
                   ->left ;
                   if (stack[sp].right->tag !
                   =--apply) break ;
               }
               switch (stack[sp].right->tag) {
                 case--int :
                 case--atom :
                 case--nilt :
                 case--colon : break ;
                 case--comb : itmp=stack[sp].right
                              ->v.ival ;
                              SuperComb[itmp] ( ) ;
                              break;
               default : itmp=stack[sp].right->tag ;
                         Comb[itmp] ( ) ;
               }
               break
case -comb : /*COMB 0 C*/
             if(SuperArgs[stack[sp].right->v.ival]=
             = 0)
               SuperComb[stack[sp].right - > v.ival]
             ( ) ;
             else { }/*COMB K C*/
             break ;
case--int :
case--atom :
case--nilt :
case--colon :
default : /*built-in Comb*/
          break;
}
 *GET*/
if(stack[sp].right==NULL)
  stack[sp].tag=--nilt ;
else {
  if(stack[sp].right->tag==--int)
    stack[sp].v.ival=stack[sp].right->v.ival ;
  else strcpy(stack[sp].v.cval,stack[sp].right->v.
  cval) ;
  stack[sp].tag=stack[sp].right->tag ;
  stack[sp].right=NULL ;
}
/*PUSH 3*/
...
/*PUSH 3*/
...
/*EVAL*/
...
/*GET*/
...
/*ADD*/
stack[sp-1].v.ival = stack[sp].v.ival + stack[sp-1].v.
ival ;
sp--;
/*UPDINT 5*/
...
/*POP 4*/
sp-=4 ;
/*RETURN*/
return ;
}
```

(그림 15) C_0에 대한 G-기계어 루틴에 대한 C 함수
(Fig. 15) The C function for the G-code routine 'C_0

5. 실험 및 분석

번역기는 UNIX 환경에서 Lex, YACC와 C로 구현하였으며 전체 번역기 소스는 약 6,000여행이다. 번역기의 정확성을 검토하기 위해서 다수의 시험용 프로그램을 작성하여 각 단계의 번역결과와 실행결과를 분석하여 검토하였으며, 실험결과로서 부록 3의 시험프로그램의 수행속도를 정리하여 나타내면 〈표 2〉와 같다. 〈표 2〉에서 시험프로그램의 수행속도는 주기억이 64 Megabyte인 SUN SPARC 10에서 30번의 반복수행으로 얻은 결과의 평균이다. 시험프로그램 Ackermann은 잘 알려진 Ackermann 함수를 구현한 것으로 실인수로 (3, 3)을 적용한 것이고, Dacsum은 divide and conquer 방식으로 합을 구하는 프로그램으로 실인수로 (1, 10000)을 적용한 것이고, Fibonacci는 Fibonacci 수를 구하는 프로그램으로 실인수로 (0, 1, 20)을 적용한 것이고, Hanoi는 하노이의 탑 알고리즘을 구현한 것으로 실인수 (1, 2, 3, 10)을 적용한 것이고, Hosum은 고계함수를 사용한 프로그램의 예로 실인수 10000을 적용한 것이고, Tak은 재귀호출이 아주 많은 프로그램으로서 실인수로 (18, 10, 6)을 적용한 것이다.

〈표 2〉 예 프로그램의 수행속도
〈Table 2〉 Execution time for example programs

Test Program	Execution speed(secs)			
	Average	Maximum	Minimum	Variance
Ackermann	1.231	1.373	1.142	0.0047
Dacsum	2.939	3.363	2.680	0.0444
Fibonacci	0.008	0.008	0.007	0.0000
Hanoi	2.685	2.965	2.506	0.0217
Hosum	3.605	5.256	3.226	0.1798
Tak	5.493	5.962	5.222	0.0507

6. 결 론

함수언어의 수행속도를 개선하려면 함수프로그램의 수행을 기계어 수준으로 구현하여야 한다. 본 연구에서는 이를 위하여 지연어의를 갖는 원시함수언어를 정의하였고 그것의 번역기를 개발하였다. 번역기의 실행모형은 G-기계를 기반

으로 한 컴비네이터 그래프축소이며, 목적어는 C를 사용하였으며, 번역결과인 C 프로그램을 C 컴파일러로 번역하여 실행 프로그램을 얻었다. 번역기는 전체 4단계로 구성되며 UNIX 환경에서 C로 작성되었다. 번역기의 정확성 검토를 위해서 다수의 시험 프로그램을 작성하여 이들을 번역하여 각 단계별 번역결과와 실행결과를 분석하여 정확함을 보았다. 향후 연구과제로는 원시언어에 입출력 기능, 패턴매칭 등의 구문을 추가하는 것에 관한 연구, 스트릭트니스 분석(strictness analysis)[3] 등의 프로그램 정적 분석을 통하여 보다 효율적인 G-코드 번역기법에 대한 연구와 효율적인 메모리 관리 기법에 관한 연구 등이다.

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, 'The implementation of the functional programming language', Prentice-Hall International, 1987.

[4] 윤영우, '요구구동형 병렬처리시스템의 설계 및 구현', 영남대학교 박사학위 논문, 1988.

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

[6] T. Johnsson, "Lambda lifting:transforming programs to recursive equations", Springer-Verlag LNCS 201, pp. 191-204, 1985.

[7] T. Johnsson, "Target code generation from G-machine code", Springer-Verlag LNCS 279, pp. 119-159, 1987.

[8] L. Augustsson, "A compiler for lazy ML", ACM LFP'84, pp. 218-227, 1984.

[9] D. A. Turner, "A new implementation techniques for the applicative language", Software-practice and experience, Vol. 9, pp. 31-49, 1979.

[10] R. J. M. Hughes, "Super-combinator:A new implementation method for applicative language", ACM LFP'82, pp. 253-264, 1982.

[11] J. Fairbairn, S. Wray, "TIM-a simple lazy abstract machine to execute supercombinators", Springer-Verlag LNCS 274, pp. 34-45, 1987.

[12] G. L. Burn, S. L. Peyton Jones, J. Robson, "The spineless G-machine", ACM LFP'88, pp. 244-258, 1988.

[13] G. Cousineau, P. L. Curien, M. Mauny, "The categorical Abstract Machine", Springer-Verlag LNCS 201, pp. 50-64, 1985.

[14] S. L. Peyton Jones, J. Salkild, "The spineless tagless G-machine", FPCA'89 Conference Proc., pp. 184-201, 1989.

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

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

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

[18] A. V. Aho, R. Sethi, J. D. Ullman, 'Compilers:Principles, Techniques, and Tools', Addison-wesley, 1986.

[19] 오세만, '컴파일러 입문', 정익사, 1988.

[20] 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.

[21] 최관덕, 문성헌, 윤영우, "함수언어 SubMiranda의 Serial Combinator 번역기 구현", 한국정보과학회 '90 가을 학술발표논문집,

pp. 321-324, 1990.

[22] 정성윤, 최관덕, 윤영우, 강병욱, "순차 컴퓨터를 기반으로하는 병렬 그래프 리덕션", 1991년도 대한전자공학회 추계종합학술대회 논문집. pp. 155-157, 1991.

Appendix 1. G-machine state transition rules

1) <O, BEGIN : C, S, G, D> ⇒ <O, C, [ ], [ ], [ ]>
2) <O, PRINT : C, n : S, G[n = INT i], D> ⇒ <O : i, C, S, G, D>
3) <O, PRINT : C, n : S, G[n=ATOM a], D> ⇒ <O : a, C, S, G, D>
4) <O, PRINT : C, n : S, G[n=CONS n_1 n_2], D> ⇒ <O, EVAL : PRINT : EVAL:PRINT : C, n_1 : n_2 : S, G, D>
5) <O, EVAL : C, v : S, G[v = AP v' n], D> ⇒ <O, UNWIND : [ ], v:[ ], G, (S, C) : D>
6) <O, EVAL : C, n : S, G[n=COMB 0 C'], D> ⇒ <O, C' : [ ], n : [ ], G, (S, C) : D>
7) <O, EVAL : C, n : S, G[n=INT i], D> ⇒ <O, C, n : S, G, D>
8) <O, EVAL : C, n : S, G[n=ATOM a], D> ⇒ <O, C, n : S, G, D>
9) <O, EVAL : C, n : S, G[n = CONS n_1 n_2], D> ⇒ <O, C, n : S, G, D>
10) <O, EVAL : C, n : S, G[n = COMB K C], D> ⇒ <O, C, n : S, G, D>
11) <O, UNWIND : [ ], n : [ ], G[n = INT i], (S,C) : D> ⇒ <O, C, n : S, G, D>
12) <O, UNWIND : [ ], n : [ ], G[n = ATOM a], (S, C) : D> ⇒ <O, C, n : S, G, D>
13) <O, UNWIND : [ ], n : [ ], G[n=CONS n_1 n_2], (S, C) : D> ⇒ <O, C, n : S, G, D>
14) <O, UNWIND:[ ], v : S, G[v = AP v' n], D> ⇒ <O, UNWIND : [ ], v' : v : S, G, D>
15) <O, UNWIND:[ ], v_0 : v_1:... : v_k : S, G[v_0 = COMB k C, v_i = AP v_{i-1} n_i ($1 \le i \le k$)], D> ⇒ <O, C, n_1:n_2:...:n_k:v_k:S, G, D>
16) <O, UNWIND:[ ], v_0:v_1:...:v_a : [ ], G[v_0 = COMB k C'], (S,C) : D> {a<k} ⇒ <O, v_a : S, G, C, D>
17) <O, RETURN : [ ], v_0:v_1:...:v_k:[ ], G, (S,C) : D> ⇒ <O, C, v_k : S, G, D>
18) <O, JUMP L:... LABEL L : C, S, G, D> ⇒ <O, C, S, G, D>
19) <O, JFALSE L : C, true : S, G, D> ⇒ <O, C, S, G, D>
20) <O, JFALSE L:... LABEL L : C, false : S, G, D> ⇒ <O, C, S, G, D>
21) <O, PUSH k : C, n_0 : n_1 : ... : n_k : S, G, D> ⇒ <O, C, n_k : n_0 : n_1:...: n_k : S, G, D>
22) <O, PUSHINT i : C, S, G, D> ⇒ <O, C, n : S, G[n = INT i], D>
23) <O, PUSHATOM a : C, S, G, D> ⇒ <O, C, n : S, G [n = ATOM a], D>
24) <O, PUSHGLOBAL f : C, S, G, D> ⇒ <O, C, n : S, G[n = COMB f], D>
25) <O, PUSHBASIC i : C, S, G, D> ⇒ <O, C, i : S, G, D>
26) <O, PUSHBATOM a : C, S, G, D> ⇒ <O, C, a : S, G, D>
27) <O, POP k : C, n_1:n_2:...: n_k : S, G, D> ⇒ <O, C, S, G, D>
28) <O, UPDATE k : C, n_0 : n_1:...: n_k : S, G, D> ⇒ <O, C, n_1:...: n_k: S, G[n_k=G n_0], D>
29) <O, UPDINT k : C, i : n_1:...: n_k : S, G, D> ⇒ <O, C, n_1:...: n_k : S, G[n_k=INT i], D>
30) <O, UPDATOM k : C, a : n_1:...: n_k : S, G, D> ⇒ <O, C, n_1:...: n_k : S, G[n_k=ATOM a], D>
31) <O, ALLOC k : C, S, G, D> ⇒ <O, C, n_1 : n_2:...: n_k : S, G[n_1 = HOLE, ..., n_k = HOLE], D>
32) <O, HEAD : C, n : S, G[n = CONS n_1 n_2], D> ⇒ <O, C, n_1 : S, G, D>
33) <O, TAIL : C, n : S, G[n = CONS n_1 n_2], D> ⇒ <O, C, n_2 : S, G, D>
34) <O, NOT : C, i : S, G, D> ⇒ <O, C, (!i) : S, G, D>
35) <O, ADD : C, i_1 : i_2 : S, G, D> ⇒ <O, C, (i_1 + i_2) : S, G, D>
36) <O, SUB : C, i_1 : i_2 : S, G, D> ⇒ <O, C, (i_1−i_2) : S, G, D>
37) <O, MULT : C, i_1 : i_2 : S, G, D> ⇒ <O, C, (i_1*i_2) : S, G, D>
38) <O, DIV : C, i_1 : i_2 : S, G, D> ⇒ <O, C, (i_1/i_2) : S, G, D>
39) <O, EE : C, i_1 : i_2 : S, G, D> ⇒ <O, C, (i_1==i_2) : S, G, D>
40) <O, NE : C, i_1 : i_2 : S, G, D> ⇒ <O, C, (i_1!=i_2) : S, G, D>
41) <O, GT : C, i_1 : i_2 : S, G, D> ⇒ <O, C, (i_1>i_2) : S, G, D>
42) <O, GE : C, i_1 : i_2 : S, G, D> ⇒ <O, C, (i_1>=i_2) : S, G, D>
43) <O, LT : C, i_1 : i_2 : S, G, D> ⇒ <O, C, (i_1<i_2) : S, G, D>
44) <O, LE : C, i_1 : i_2 : S, G, D> ⇒ <O, C, (i_1<=i_2) : S, G, D>
45) <O, OR : C, i_1 : i_2 : S, G, D> ⇒ <O, C, (i_1 | | i_2) : S, G, D>
46) <O, AND : C, i_1 : i_2 : S, G, D> ⇒ <O, C, (i_1&&i_2) : S, G, D>
47) <O, MKAP : C, n_1 : n_2 : S, G, D> ⇒ <O, C, n : S, G [n = AP n_2 n_2], D>
48) <O, MKAP K : C, n_0 : n_1:...: n_k : S, G, D> ⇒ <O, C, v_0 : v_1:...: v_k : S, G[v_0 = AP n_1~n_2, v_i = AP v_{i-1} n_{i+1}, ($1 \le i \le k$)], D>
49) <O, MKINT : C, i : S, G, D> ⇒ <O, C, n : S, G[n = INT i], D>
50) <O, CONS : C, n_1 : n_2 : S, G, D> ⇒ <O, C, n : S, G[n =CONS n_1 n_2], D>
51) <O, GET : C, n : S, G[n = INT i], D> ⇒ <O, C, i : S, G, D>
52) <O, GET : C, n : S, G[n = ATOM a], D> ⇒ <O, C, a : S, G, D>

Appendix 2. G-machine compilation rules

In the following rules, ρ is a function which takes an identifier and returns a number giving the offset of the correnponding argument from the base of current context and d is the depth of the current context minus one. f denotes program-drived or built-in supercombinator, x denotes formal parameter or local variables, and E denotes a curried expression.

F[SCDef] generates code for a supercombinator definition SCDef.

F[f x1 x2··· xn = E] = GLOBSTART f n; R[E] [x1= n, x2=n−1, ..., xn=1] n

---

R[E] ρ d generates code to applay a supercombinator to its d arguments.

R[i] ρ d=B[i] ρ d ; UPDINT (d+1) ; POP d ; RETURN
R[a] ρ d = B[a] ρ d ; UPDATOM (d+1) ; POP d ; R-ETURN

```
R[f] ρ d = E[f] ρ d ; UPDATE (d-1) ; POP d ; UNWIND
R[x] ρ d = E[x] ρ d ; UPDATE (d-1) ; POP d ; UNWIND
R[HEAD E] ρ d = E[HEAD E] ρ d ; UPDATE
    (d-1) ; POP d ; RETURN
R[TAIL E] ρ d = E[TAIL E] ρ d ; UPDATE (d-1) ; POP
    d ; RETURN
R[~E] ρ d = B[~ E] ρ d ; UPDINT (d-1) ; POP
    d ; RETURN
R[- E1 E2] ρ d = B[+ E1 E2] ρ d ; UPDINT
    (d-1) ; POP d ; RETURN
    similary for -, *, /, and, or,
    ==, !=, >, <, >= and <=.
R[CONS E1 E2]ρ d = E[CONS E1 E2]ρ d ; UPDATE
    (d-1) ; POP d ; RETURN
R[IF Ec Et Ef] ρ d = B[Ec]ρ d ; JFALSE L ; R[Et]ρ d ;
RETURN ; LABEL L ; R[Ef]ρ d
R[E1 E2]ρ d = RS[E1 E2] ρ d 0 ;
R[letrec D in E]ρ d = C[D]ρ' d' ; R[E]ρ' d' ;
    where (ρ', d') = X[D]ρ d
```

RS[E]ρ d n completes a supercombinator reduction, in which the top n ribs of the body have already been put on the stack. RS constructs instances of the ribs of E, putting them on the stack, and then completes the reduction in the same way as R.

```
RS[f]ρ d n = PUSHGLOBAL f ; MKAP n ; UPDATE
    (d-n-1) ; POP (d-n) ; UNWIND
RS[x]ρ d n = PUSH (d-ρ x) ; MKAP n ; UPDATE
    (d-n-1) ; POP (d-n) ; UNWIND
RS[HEAD E]ρ d n = E[E]ρ d ; MKAP n ; UPDATE
    (d-n-1) ; POP (d-n) ; UNWIND
RS[TAIL E]ρ d n = E[E]ρ d ; MKAP n ; UPDATE
    (d-n+1) ; POP (d-n) ; UNWIND
RS[IF Ec Et Ef]ρ d n = B[Ec]ρ d ; JFALSE L1 ; RS[Et]ρ
    d n ; JUMP L2 ; LABEL L1 ; RS[Ef]ρ d n ; LABEL L2
RS[IF E]ρ d n = B[E]ρ d ; JFALSE L1 ; PUSHGLOBAL
    K-2-1 ; UPDATE (d+1) ; POP d ; UNWIND ; JUMP
    L2 ; LABEL L1 ;
    PUSHGLOBAL K-2-2 ; UPDATE (d+1) ; POP d ;
    UNWIND ; LABEL L2
RS[E1 E2]ρ d n = C[E2]ρ d ; RS[E1]ρ (d+1) (n+1)
```

E[E]ρ d evaluates E, leaving the result on top of the stack.

```
E[i]ρ d = PUSHINT i
E[a]ρ d = PUSHATOM a
E[f]ρ d = PUSHGLOBAL f ; EVAL
E[x]ρ d = PUSH(d-ρ x) ; EVAL
E[HEAD E]ρ d = E[E]ρ d ; HEAD ; EVAL
E[TAIL E]ρ d = E[E]ρ d ; TAIL ; EVAL
E[~E]ρ d = B[~E]ρ d ; MKINT
E[+E1 E2]ρ d = B[+E1 E2] ρ d ; MKINT
    similary for -, *, /, and, or, ==, !=, >,
    <, >= and <=.
E[CONS E1 E2]ρ d = C[E2]ρ d ; C[E1]ρ (d+1) ; CONS
E[IF Ec Et Ef]ρ d = B[Ec]ρ d ; JFALSE L1 ; E[Et]ρ d ;
    JUMP L2 ; LABEL L1 ; E[Ef]ρ d ; LABEL L2
    E[E1 E2]ρ d = ES[E1 E2]ρ d 0
```

ES[E]ρ d n completes the evaluation of an expression, the top n ribs of which have already been put on the stack. ES constructs instances of the ribs of E, putting them on the stack and then completes the evaluation in the same ways as E.

```
ES[f]ρ d n = PUSHGLOBAL f ; MKAP n ; EVAL
ES[x]ρ d n = PUSH (d - ρ x) ; MKAP n ; EVAL
ES[HEAD E]ρ d n=E[E]ρ d ; HEAD ; MKAP n ; EVALES
[TAIL E]ρ d n=E[E]ρ d ; TAIL ; MKAP n ; EVALES
[IF Ec Et Ef]ρ d n = B[Ec]ρ d ; JFALSE L1 ; ES[Et]ρ
    d n ; JUMP L2 ; LABEL L1 ; ES[Ef]ρ d n ; LABEL L2
ES[IF E]ρ d n = B[E]ρ d ; JFALSE L1 ; PUSHGLOBAL
    K-2-1 ; UPDATE (d+1) ; POP d ; UNWIND ; JUMP
    L2 ; LABEL L1 ; PUSHGLOBAL K-2-2 ; UPDATE
    (d+1) ; POP d ; UNWIND ; LABEL L2
ES[E E1]ρ d n = C[E1]ρ d ; ES[E]ρ (d-1) (n+1)
```

B[E]ρ d evaluates E, leaving the result on top of the stack as a naked basic value.

```
B[i]ρ d = PUSHBASIC i
B[a]ρ d = PUSHBTOM a
B[~ E]ρ d = B[E]ρ d; NOT
B[-E E1]ρ d = B[E1]ρ d ; B[E1]ρ (d-1) ; ADD
    similary for -, *, /, and, or, ==, !=, >,
    <, >= and <=.
B[IF E E1 E2]ρ d = B[Ec]ρ d ; JFALSE L1 ; B[Et]ρ
    d ; JUMP L2 ;
    LABEL L1 ; B[E]ρ d ; LABEL L2
B[E]ρ d = E[E]ρ d ; GET
```

C[E]ρ d constructs the graph for an instance of E in a context given by ρ and d. It leaves a pointer to the graph on top of the stack.

```
C[i]ρ d = PUSHINT i
C[a]ρ d = PUSHATOM a
C[f]ρ d = PUSHGLOBAL f
C[x]ρ d = PUSH (d - ρ x)
C[E E1]ρ d = C[E1]ρ d ; C[E1]ρ (d-1) ; MKAP
```

C[D]ρ d takes mutually recursive set of definitions D, constructs an instance of each body, and leaves the pointers to the instances on top of the stack.

```
C[x1=E1 ... xn=En]ρ d = ALLOC n ; C[E1]ρ d ; UPDATE
n ; ... C[En]ρ d ; UPDATE 1 ;
```

X[D]ρ d returns a pair (ρ', d') which gives the context augmented by the definitions D.

```
X[x1=E1 ... xn=En]ρ d = (ρ[x1=d+1 ... xn=d-n], d-n)
```

Appendix 3. Test programs

1) Ackermann

```
{ackermann m n = n-1, m==0
               = ackermann (m-1 1), n==0
               = ackermann (m-1 ackermann (m n-1))
} ackermann (3 3)
```

2) Dacsum

```
{sum low high = low, high==low
              = low+high, high==low+1
              = sum(low mid) + sum(mid+1 high)
                where { mid=(high+low)/2 }
} sum (1 10000)
```

3) Fibonacci

```
{fib x y n = y, n==0
           = fib (y x+y n-1)
} fib (0 1 20)
```

4) Hanoi

```
{hanoi f s t n = move (f s), n==1
               = append (hanoi(f t s n-1)
                 move(f s) : hanoi(t s f n-1))
                 where{
                 append (a:x) y = a:y, x==null
                                = a : append(x y)
                 move x y = (10*x)+ y
                 }
} hanoi (1 2 3 10)
```

5) Hosum

```
{sumints m = sum(map(addone count(1)))
           where
           {
```

```
count n = null, n>m
        = n:count(n+1)
map f (l : ls)=null, l==null
              = f(l):map(f ls)
                addone x = x+1
                }
sum (l:ls) = 0, l==null
           = l, ls==null
           = l + sum(ls)
  } sumInts (10000)
6) Tak
  {tak x y z = z, ~(y < x)
             = tak (tak(x-1 y z)
                    tak(y-1 z x)
                    tak(z-1 x y))
  } tak (18 10 6)
```


이 종 희

1978년 경북대학교 공과대학 전자계산기공학과(공학사)
1981년 영남대학교 대학원 전자공학과 계산기전공(공학석사)
1990년 영남대학교 대학원 전자공학과 계산기전공(박사과정 수료)
1978년 ~79년 (주)한국정보시스템
1984년~현재 영남전문대학 전자계산기과 교수
관심 분야:함수언어, 병렬 처리

최 관 덕

1989년 영남대학교 공과대학 전산공학과(공학사)
1991년 영남대학교 대학원 전산공학과 전산기구조전공(공학석사)
1995년 영남대학교 대학원 전산공학과 컴퓨터시스템전공(박사과정 수료)
관심분야:함수언어, 병렬처리

윤 영 우

1972년 영남대학교 공과대학 전자공학과(공학사)
1984년 영남대학교 대학원 전자공학과(공학석사)
1988년 영남대학교 대학원 전자공학과(공학박사)
1988년~현재 영남대학교 공과대학 전산공학과 부교수
관심분야:컴퓨터 구조, 프로그래밍 언어, 병렬처리

강 병 욱

1970년 영남대학교 공과대학 전자공학과(공학사)
1977년 영남대학교 대학원 전자공학과(공학석사)
1994년 경북대학교 대학원 전자공학과(공학박사)
1979년~현재 영남대학교 공과대학 전산공학과 교수
관심분야:소프트웨어 공학, 프로그래밍 언어