프로그램유도 컴비네이터를 이용하는 함수프로그램의 프로세스망 구성

신 승 철† · 유 원 희††

요 약

병렬 구문을 갖지 않는 함수 프로그램의 병렬 수행을 위해 람다 계산 해석법(λ-calculus encoding)이 도입되었고 이것은 함수 프로그램을 프로세스 계산(process calculus)을 이용하여 프로세스망으로 구성하고 프로세스간의 통신 행위에 의해 결과를 얻는 새로운 계산 모델에서 사용될 수 있다. 그러나 람다 계산 해석법은 상수식 조차도 너무 많은 통신 행위를 야기시키는 문제가 지적되어 왔다. 본 논문은 병렬 구문을 갖지 않는 컴비네이터 프로그램을 위한 해석법을 제안한다. 또 이것은 프로세스망 리덕션과 그래프 리덕션을 결합할 수 있도록 계산 프로세스인 초어 프로세스(chore process;chore)를 도입한다. 초어는 지역 그래프 리덕션이 가능한 상수식을 위한 그래프 리덕션 함수를 포함할 수 있으며 초어 프로세스의 생성은 주어진 컴비네이터 프로그램에 대한 표식과 변환에 의해 추출되는 컴비네이터 적용식을 포함하지 않는 G-감축가능한(G-reducible) 부분식으로부터 이루어진다. 본 논문은 이러한 초어 프로세스를 포함하는 해석법으로 생성된 프로세스망이 초어를 갖지 않는 것보다 더 적은 통신 행위의 회수를 발생한다는 것을 보인다.

Functional Programs as Process Networks using Program-derived Combinators

Seung Cheol Shin † · Weon Hee Yoo ††

ABSTRACT

For parallel implementations of functional programs without concurrent primitives, the λ-calculus encodings have been introduced. A functional program may be transformed into a process network using process calculi by the λ-calculus encoding and the result of a program can be obtained by a deal of communication actions in it's process network. But the λ-calculus encodings cause too many communication actions even in constant expressions. This paper shows the encoding for a combinator program without concurrency primitives which can combine the graph reduction and the process-net reduction using computable processes, 'chores'. A 'chore' may have graph reduction functions for primitive operations of constants for which local graph reduction may be possible, and be encoded from a 'G-reducible' subexpression which is obtained by an annotation and transformation for a combinator program, assuring that it does not include any combinator application. Also, we show that a process network with chores raises less communication actions than one without chores.

*본 논문의 연구는 1994년도 인하대학교 교내연구비 지원에 의하여 수행됨.
† 정 회 원:동양대학교 컴퓨터공학부
†† 정 회 원:인하대학교 전자계산공학과
논문접수:1995년 9월 5일, 심사완료:1996년 1월 24일

1. 서 론

함수언어의 중요한 특성중의 하나는 함수언어가 내재적인 병렬성을 갖는다는 것이다. Peyton Jones는

함수언어의 구현 모델로서 블로킹을 갖는 병렬 그래프 리덕션을 제안하였다[16, 17]. 이 모델은 전통적인 그래프 리덕션을 기반으로 하고 전역 그래프 공간을 채용한다. Goldberg[6]의 Alfalfa와 Buckwheat는 이러한 Peyton Jones의 모델을 구현한 예로 알려져 있다.

최근에는 전역 그래프 공간을 요구하지 않는 새로운 방법이 연구되었는데 이것은 Milner의 파이 계산(π-calculus)[14], Thomsen의 CHOCS[20], Leth의 LCCS[11]등과 같은 프로세스 계산(process calculus; process algebra)을 이용하는 것이다. 이러한 프로세스 계산들은 임의의 동적인 구조를 갖는 병렬 시스템을 표현하기위해 CCS(Calculus of Communicating System)[12]로부터 확장된 것이다. 프로세스와 함수간의 관계를 규명하는 많은 연구가 이루어지고 있으며 특히 Milner는 지연 람다 계산(lazy λ-calculus)[1]의 연산에 대한 시뮬레이션을 프로세스망으로 표현하기 위해 파이 계산을 이용한 람다 계산 해석법(λ-calculus encoding)을 발표하였다[15]. Leth[11]와 Glauert[4, 5]는 각각 Milner의 해석법을 기반으로 하는 새로운 병렬 구현 방법을 제안하였다. 본 논문은 이러한 모델을 그래프 리덕션에 반하여 프로세스망 리덕션(process-net reduction) 이라 부르기로 한다. 좀더 정확하게 말하자면 프로세스망 리덕션은 함수 프로그램을 그래프 대신에 프로세스망으로 표현하고 이 프로세스망에서의 통신 행위(communication activity)를 통하여 프로그램의 결과를 얻는 계산 모델을 의미한다.

병렬 프로그래밍 언어의 실제적인 구현을 위한 기반으로 프로세스 계산의 여러가지 개념들이 사용되어 왔으며 Facile[3, 18]과 CML[19]은 함수와 프로세스를 결합한 좋은 예이다. 그러나 Facile과 CML은 병렬구문이 프로그래머에게 제공되고 원시 프로그램이 병렬구문을 포함할 수 있다는 점에서 람다 계산 해석법을 기반으로 하는 모델과 다르다고 할 수 있다.

Leth[11]는 자신의 LCCS(Label-passing CCS)를 이용하여 무형 람다 계산, SKI 컴비네이터, Kennaway와 Sleep의 지시 스트링(director string)[10] 등의 구현 방법을 제시하였다. Glauert[5]는 튜플 값의 전달을 허용함으로써 Honda와 Tokoro의 비동기 프로세스 모델[7]을 기반으로 하는 프로세스 계산 이론에 실용적인 확장을 제시하고 상수를 포함하는 람다 계산 해석법을 제안하였다.

본 논문은 병렬구문을 갖지 않는 컴비네이터 프로그램에 대하여 그래프 리덕션과 프로세스망 리덕션을 결합하는 새로운 해석법을 제안한다. 그래프 리덕션과 프로세스망 리덕션을 결합하는 시도는 Facile의 추상기계인 C-FAM[3, 18]에서 구현되었지만 Facile 프로그램의 병렬구문은 어떤 해석법에 의해 얻어지는 것이 아니라 프로그래머에 의해서 주어진다. 프로세스망 리덕션만을 위한 기존의 람다 계산 해석법들은 너무 많은 프로세스를 발생시키고 따라서 상수식에서 조차도 너무 많은 통신 행위를 야기시킨다. 본 논문은 지역 그래프 리덕션이 가능한 상수들의 기본 연산식에 대하여는 프로세스망 리덕션 대신에 그래프 리덕션을 수행하도록 프로세스망을 구성한다.

본 논문은 몇가지 전제로부터 출발하고자 한다. 첫째, 원시 프로그램이 주어지면 지역 그래프 리덕션이 가능한 부분식들을 찾아내야 하는데 이때, 원시 프로그램은 수퍼컴비네이터[9]나 시리얼 컴비네이터(정확히 말하자면 SCIF; Serial Combinator Intermediate Form)[8]와 같은 프로그램유도 컴비네이터의 프로그램이다. 둘째, 본 논문은 프로세스망 리덕션 시간과 그래프 리덕션 시간의 비교에 대하여는 구체적인 관심을 갖지 않는다. 단지, 상수들의 기본 연산식에 대한 순차(sequential) 그래프 리덕션이 프로세스망 리덕션보다 시간 복잡도면에서 더 많은 오버헤드를 갖지 않는다고 가정한다. 특히, 시리얼 컴비네이터와 같이 컴비네이터 본체에서 병행부분식(conc-urrent subexpression)을 추출하여 추상화한 컴비네이터는 본체 내의 상수식들을 순차적으로 처리하는 것이 병렬로 처리하는 것보다 수행 시간면에서 결코 뒤지지 않는다고 알려져 있으며[6, 8], 이것으로 본 논문의 전제에 대한 이유를 설명할 수 있다.

본 논문의 이해를 위해서는 람다 계산의 리덕션 규칙, 함수 언어 구현, 프로세스 계산, 람다 계산 해석법 등의 기본 지식이 필요하지만 본문에서 직접 사용하는 개념들은 요약하여 설명하였고 그외의 자세한 사항은 참고문헌[1, 4, 5, 14, 15]에서 찾을 수 있다.

논문의 구성은 먼저, 2장에서 본문에서 필요한 프로세스 계산과 프로세스망 리덕션의 개념을 정리하고, 3장에서는 주어진 컴비네이터 프로그램으로부터 지역 그래프 리덕션을 통한 순차처리가 가능한 상수식들을 추출하는 방법을 표식(annotation)과 변환

(transformation)으로 설명한다. 4장은 변환된 컴비네이터 프로그램을 프로세스망으로 해석하는 방법을 그래프 리덕션 함수를 포함하는 초어(chore) 프로세스의 도입을 통하여 제안한다. 5장은 프로세스망 리덕션의 시뮬레이션을 통하여 기존의 프로세스망보다 초어 프로세스를 갖는 프로세스망이 리덕션 회수에 있어서 우수하다는 결과을 제시한다.

2. 프로세스 계산과 프로세스망 리덕션

병렬 시스템의 동작과 의미를 표현하기 위한 정형 시스템(formal system)으로 처음 개발된 Milner의 CCS [12, 13]는 함수 시스템(functional system)의 람다 계산과 같은 역할을 하는 이론적인 기조를 병렬시스템에 제공하기 위한 것이였지만 시스템 구조가 변하지 않는 정규 시스템(regular system)을 표현할 수 있는 반면에 동적으로 변화하고 재구성되는 병렬 시스템은 표현할 수 없다는 문제를 갖는다. 따라서 동적으로 확장가능한 프로세스망을 모델링하기 위해 CCS를 확장하는 프로세스 계산에 대한 많은 연구가 이루어져 왔으며 그 결과, Milner 자신의 파이 계산[14]과 그 아류로 간주되는 Thomsen의 CHOCS[20], Leth의 LCCS[11], Glauert의 ACPL[4] 등이 제안되었다.

한편, 이들 프로세스 계산 모델을 이용하여 함수와 프로세스의 관계를 규명하고 함수 언어의 병렬 수행 모델을 모색하는 연구도 병행되어 람다 계산의 병렬 연산에 대한 시뮬레이션을 표현하는 람다 계산 해석법이 다양하게 제안되었고 그중 Glauert[4, 5]는 파이 계산에서 람다 계산 해석을 위한 부분만을 추출하여 ACPL을 정의하고 값호출 방식과 지연 방식에 대한 일률적인 해석법을 제안함으로써 기본 연산자와 상수를 포함하는 좀더 실용적인 모델을 제안하였다.

본 논문은 Glauert의 ACPL과 람다 계산 해석법을 바탕으로 람다식이 아닌 컴비네이터식에 대한 값호출과 지연 해석법을 각각 제안하고 생성되는 프로세스망에 그래프 리덕션 함수를 삽입하여 통신 행위를 억제하는 방법을 제시한다.

먼저, Glauert의 ACPL을 중심으로 프로세스 계산을 설명한다. 본문에 걸쳐 사용되는 프로세스 계산의 구문과 시멘틱스는 ACPL과 다른 점이 없으며 프로세스 항(term)의 구문은 다음과 같이 정리할 수 있다.

$$P ::= x!y \mid x?y.P \mid P\backslash x \mid (P \mid Q) \mid P : A(x_1,\ldots,x_n) = Q \mid A(y_1,\ldots,y_n)$$

P와 Q는 프로세스(또는 agent라고도 한다.)이고 A는 프로세스 명칭이며, x는 프로세스간의 링크이고 y는 값이나 링크 명칭을 나타낸다. 위의 각 구문은 링크 x를 통한 출력과 입력, 링크 명칭의 영역 제한(restriction), 병렬 조합(parallel composition), 프로세스 정의, 정의

〈표 1〉 프로세스 계산의 전이 규칙

〈Table 1〉 Transition rules of process calculus

Output: $x!y.P \xrightarrow{x!y} P$	Input: $x?y.P \xrightarrow{x?w} P[w/y]$
Restriction: $\dfrac{P \xrightarrow{\alpha} P'}{P\backslash x \xrightarrow{\alpha} P'\backslash x}, x \neq c(\alpha)$	Open restriction: $\dfrac{P \xrightarrow{y!x} P'}{P\backslash x \xrightarrow{y\backslash z} P'[z\backslash x]}, z \notin FN(P\backslash x)$
Close restriction: $\dfrac{P \xrightarrow{x?y} P' \quad Q \xrightarrow{x\backslash y} Q'}{P \mid Q \xrightarrow{\tau} (P' \mid Q')\backslash y}$	Communication: $\dfrac{P \xrightarrow{x?y} P' \quad Q \xrightarrow{x!y} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$
Parallel composition: $\dfrac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}, BN(\alpha) \cap FN(Q) = \phi$	Agents: if $A(x_1,\ldots,x_n = P$, $\dfrac{P[y_1/x_1,\ldots,y_n/x_n] \xrightarrow{\alpha} P'}{A(y_1,\ldots,y_n) \xrightarrow{\alpha} P'}$

된 프로세스의 발생 등을 각각 나타낸다. 이때 출력항은 Honda와 Tokoro의 모델[7]과 같이 비동기 출력을 의미한다. 또한 $x!(u,v) \mid x?(u, v). P$와 같이 튜플의 출력과 입력이 간섭(interleaving)없이 허용된다.

이와 같은 구문으로 표현되는 프로세스 망의 동작은 레이블을 갖는 전이 시스템(labelled transition system)에 의해서 그 시멘틱스를 표현할 수 있다. 전이의 레이블로 표현되는 행위 α는 입력 $x?y$, 출력 $x!z$, 제한 명칭(restricted name)의 출력 $x \backslash y$, 내부 행위(internal action) τ등 네 종류로 나누어지고 $c(\alpha)$는 행위 α의 통신 링크를 나타낸다. ?와 \는 모두 명칭 바인더(name binder)로 작용하고 BN과 FN은 각각 컴비네이터식이나 프로세스항 또는 행위에 대한 바운드 명칭과 자유 명칭을 나타낸다. 프로세스 계산의 전이 관계는 표 1의 규칙들을 만족하는 최소 관계로 정의된다.

함수 언어의 기조가 되는 람다 계산의 연산과정을 프로세스 계산의 동작으로 표현하는 람다 계산 해석법은 Milner에 의해 처음 고안되었으며[15], Glauert는 튜플값의 전달과 비동기 출력을 이용하여 Milner의 해석법을 보다 단순화시켰다[4]. 본 논문이 제시하는 컴비네이터 해석법은 그래프 리덕션과 프로세스망 리덕션의 결합을 가능하게 하는데, 이러한 연구의 동기가 된 Glauert의 값호출 람다 계산 해석법은 다음과 같다.

$$[x]u = \mathrm{rec}\ X.\ u?v.\ (x!v \mid X)$$
$$[K]u = \mathrm{rec}\ X.\ u?v.\ (v!K \mid X)$$
$$[\lambda x.M]u = \mathrm{rec}\ X.\ u?(x, q).\ ([M]q \mid X)$$
$$[MN]u = ([M]f \mid f!(a, u) \mid [N]a) \backslash f \backslash a$$

이 변환 방법은 주어진 람다식의 값호출 연산을 표현하는 프로세스망을 생성하고, 이때 rec X. P와 같은 형태는 재귀 프로세스 $X{:}X = P$를 의미한다. Glauert는 곱셈과 같은 기본 연산을 다음과 같이 미리 정의된다고 하였다.

$$[\mathrm{Mult}(M, N)]u = ([M]a \mid [N]b \mid (a!x \mid b!y \mid x?m.y?n.\ \mathrm{Const}(u, m*n)) \backslash x \backslash y) \backslash a \backslash b \qquad (1)$$

where $\mathrm{Const}(z, c) = z?r.(r!c \mid \mathrm{Const}(z, c))$.

여기서 재귀 프로세스 Const는 기본 연산의 결과를 재귀적으로 반복 출력할 수 있도록 정의된다. 이와 같은 방법으로 식(2)와 같은 람다식은 식(3)과 같이 변환될 수 있고 프로세스식(3)은 프로세스 $(u!p \mid p?v. P(v))$와 병렬 조합되어 통신 전이 관계인 리덕션에 의해서 식(2)의 결과를 얻어낼 수 있다. 이때 $P(v)$는 v값을 출력하는 프로시쥬어로 간주된다.

$$(\lambda x.\ \lambda y.\ * \ x\ y)\ 2\ 3 \qquad (2)$$

$$\begin{aligned}&[(\lambda x.\ \lambda y.\ * \ x\ y)\ 2\ 3]u \qquad (3)\\ &= (\mathrm{rec}\ X.\ f?(x, q).\ (\mathrm{rec}\ Y.\ q?(y, r).\ ((\mathrm{rec}\ X.\ c?v.\\ &\quad (x!v \mid X) \mid \mathrm{rec}\ X.\ d?v.\ (y!v \mid X)\\ &\quad \mid (c!s \mid d!t \mid s?m.t?n.\ \mathrm{Const}(r, m*n)) \backslash s \backslash t) \backslash c \backslash d \mid Y)\\ &\quad \mid X) \mid f!(a, w) \mid \mathrm{rec}\ X.\ a?v.\ (v!2 \mid X)\\ &\quad \mid w!(b, u) \mid \mathrm{rec}\ X.\ b?v.\ (v!3 \mid X)) \backslash f \backslash w \backslash a \backslash b\end{aligned}$$

3. 프로세스망과 그래프의 결합을 위한 변환

3.1 프로세스망 리덕션과 그래프 리덕션의 결합

Glauert의 해석법은 Milner의 해석법을 튜플의 전달과 비동기식 통신, 상수및 기본연산자의 정의 등을 통하여 좀더 단순하고 실용적인 프로세스망 리덕션 모델로 전개하였다고 인정되지만 여전히 주어진 람다식에 대하여 너무 많은 프로세스를 생성하고 너무 많은 통신 행위를 발생시키는 문제를 갖는다.

본 논문은 이러한 문제의 해결책을 식(4)와 같이 기본연산자가 복합적으로 적용된 예에서 설명한다. 식(4)는 곱셈 연산자나 덧셈 연산자가 미리 정의될 수 있기 때문에 프로세스식(5)와 같이 해석될 수 있다.

$$(+\ (*M\ N)\ 99) \qquad (4)$$

$$\begin{aligned}&[+\ (*\ M\ N)\ 99]u \qquad (5)\\ &= (([M]c \mid [N]d \mid (c!v \mid d!w \mid v?m.w?n.\\ &\quad \mathrm{Const}(u, m*n)) \backslash v \backslash w) \backslash c \backslash d \mid [99]b\\ &\quad \mid (a!x \mid b!y \mid x?s.\ y?t.\ \mathrm{Const}(u, s+t)) \backslash x \backslash y) \backslash a \backslash b\end{aligned}$$

이때 우리는 다음과 같은 그래프 리덕션 함수 GR을 정의한다.

[정의 3.1] G-감축가능성과 그래프 리덕션 함수 GR

(i) 람다식 E가 델타 규칙(δ-rule)을 포함하는 람다 모델 $\Lambda_\delta[\ ]$의 원소이고, 자유변수와 람다추상형을 포함하지 않는다면 E는 G-감축가능(G-reducible)하다고 한다.

(ii) G-감축가능한 람다식 E에 대하여
$GR(E) = 〔E〕$

이다. 이때 ⟦_⟧는 람다식에 대한 시멘틱 함수 $⟦_⟧_\sigma$에서 환경(environment) σ를 생략한 것이다. 변수와 람다추상형을 포함하지 않고, 기본연산자와 상수로만 이루어진 G-감축가능한 람다식은 별도의 환경없이 시멘틱 값을 얻을 수 있다.

이제 식(4)가 G-감축가능하다는 가정하에 그래프 리덕션 함수 GR을 이용하여 식(5)와 다른 새로운 해석을 식(6)으로 나타낼 수 있다.

$$[+(*M\ N)\ 99]u = ([M]a \mid [N]b \mid (a!x \mid b!y \mid x?m.\ y?n.\ u!GR(m*n+99)) \backslash x \backslash y) \backslash a \backslash b \qquad (6)$$

여기서 GR은 프로세스 계산의 어떤 프로세스 명칭이 아니라 함수임에 주의할 필요가 있다. 따라서 식(6)에서 u!GR(m*n + 52)은 m과 n에 바인드된 상수 K_1과 K_2에 대하여 상수 $K = ⟦K_1 * K_2 + 99⟧$를 출력하는 u!K가 된다.

이러한 방법은 람다계산에 대한 람다리프팅과도 유사할 뿐 아니라 기본연산자에 대한 프로세스를 미리 정의하는 방법을 복합 연산에 대한 프로세스도 미리 정의할 수 있도록 일반화한 것으로 볼 수 있다. 사실, 여기서 이용한 그래프 리덕션 함수는 기본 연산자와 상수로만 구성된 복합 상수식을 연산하는 어떤 다른 함수, 예를 들면 스택 계산 함수(stack machine)로도 대치될 수 있다. 본 논문이 GR을 선택한 이유는 어떤 평가에 의한 것이 아니라 기존의 함수언어의 계산 모델로 이용되어 왔을 뿐 아니라 간략하게 정의할 수 있다는 점 때문이다.

식(6)에서 GR의 결과값은 어떤 다른 프로세스에 의해서도 공유될 수 없지만 프로세스식의 설명이 복잡해지는 것을 막기 위해 GR의 공유문제는 4.4절에서 자세히 다루기로 하다.

3.2 컴비네이터 프로그램의 변환을 위한 표식

이제 본 논문에서 다루는 해석법의 대상인 컴비네이터식이 주어지면 앞절의 식(6)과 같은 해석이 가능한 G-감축가능한 부분식을 찾아내야 한다. 그러나 컴비네이터식은 람다추상형이나 자유변수가 이미 존재하지 않으므로 본 논문의 목적에 맞도록 컴비네이터식에 대한 G-감축가능성을 다시 정의할 필요가 있다. 또한 우리는 주어진 컴비네이터식에 대한 그래프에 독특한 표식을 줌으로써 어떤 부분식이 G-감축가능한 지를 표현한다. 이때 컴비네이터식의 형(type) 정보와 G-표현가능성 표식을 이용하여 컴비네이터식의 각 구문에 따라 G-감축가능성 표식을 정의할 수 있다. 각 부분식에 대한 형 정보는 그것이 함수형(functional type)인지 아닌지만을 구분하는 데에 사용되며 고계(higher order) 함수를 처리하는 형 추론(type inference) 시스템[16]으로부터 쉽게 얻을 수 있다.

[정의 3.2] G-감축가능성과 표식

(i) 주어진 컴비네이터식의 어떤 부분식이 함수형이 아니고 내부에 컴비네이터 적용식을 포함하지 않을 때 그 부분식은 G-감축가능하다.

(ii) 컴비네이터식에 대한 표식
- 상수 K와 기본 연산자는 G-표현가능하다.
- 변수 x는 함수형이 아니면 G-표현가능하다.
- 컴비네이터 $는 G-표현가능하지 않다.
- 적용식(application expression) MN은 M과 N이 모두 G-표현가능할 때 G-표현가능하다.
- G-표현가능한 적용식 MN은 함수형이 아닐 때 G-감축가능하다.

컴비네이터식에 대한 G-감축가능성 정의는 4.1절에서 설명될 해석법에서 그래프 리덕션 함수 GR을 적용할 부분식을 식별한다. 정의 3.2의 표식에 의해 주어진 컴비네이터식의 모든 부분식에 대하여 G-감축가능성을 표현할 수 있으며 G-감축가능한 모든 부분식은 부분 적용식(partial application)이 아니고 내부에 컴비네이터 적용식을 포함하지도 않는다. 이때 기본 연산자는 자신의 모든 인자에 대하여 스트릭트(strict)한 일계(first-order) 함수들과 넌스트릭트(nonstrict) 함수인 IF를 포함한다.

이제 주어진 컴비네이터식에 대한 표식들을 각 경우에 따라 4가지 형태로 분류한다. 표식에 대한 분류는 각 부분식들의 성질을 명확하게 하기 위한 것으로 다음 단계의 프로그램 변환에 대한 설명을 쉽게 해준다.

[정의 3.3] 표식의 분류

형태 1. G-표현가능하고 함수형인 경우
- 기본 연산자
- 컴비네이터 적용식을 포함하지 않는 기본 연산자의 부분 적용식

형태 2. G-표현가능하고 함수형이 아닌 경우(이 경우는 G-감축가능하다)

- 상수, 변수
- 컴비네이터를 포함하지 않는 기본연산자의 완전 적용식(complete application)

형태 3. G-표현가능하지 않지만 함수형인 경우

- 함수형 변수와 이에 대한 적용식
- 컴비네이터와 이에 대한 부분 적용식
- 함수형인 컴비네이터 완전 적용식
- 컴비네이터 적용식을 포함하는 기본 연산자의 부분 적용식

형태 4. G-표현가능하지않고 함수형도 아닌 경우

- 함수형이 아닌 컴비네이터 완전 적용식
- 컴비네이터 적용식을 포함하는 기본 연산자의 완전 적용식

[예 3.1] 주어진 컴비네이터식 E에 대하여 각 부분식의 표식은 다음과 같이 분류된다.

E = + (*x (+x y)) ($ x (f x y))

형태 1: +, *, + (* x (+ x y)), * x, + x

형태 2: x, y, (* x (+ x y)), (+ x y)

형태 3: f, f x, (f x y), $, $ x

형태 4: $ x (f x y), + (* x (+ x y)) ($ x (f x y))

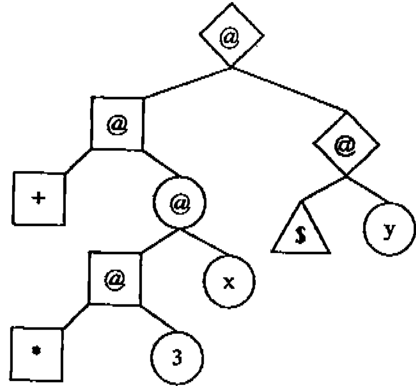
여기서 함수형 변수 'f'는 알려지지 않은 상태이므로 부분식 (f x y)이 함수형이 아닐 수도 있다. 그러나 'f'에 바인드되는 함수는 프로세스망 구성과정에서 통신 링크 'f'를 통해 연결되므로 표식 형태 3으로 분류한다는 사실에 유의할 필요가 있다.

3.3 컴비네이터 프로그램의 변환

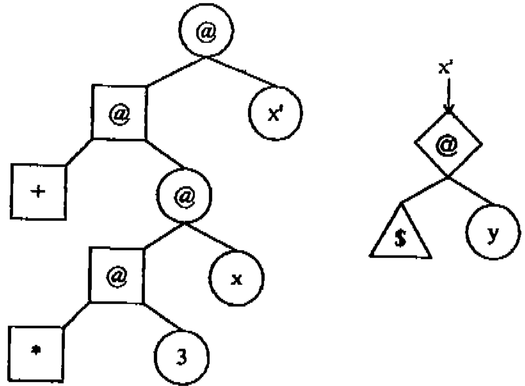
표식 분류의 4가지 형태중에서 형태 2가 G-감축가능함을 나타낸다. 이제 주어진 컴비네이터식이 G-감축가능한 부분식을 더 많이 포함하도록 컴비네이터식을 변환하는 방법을 설명한다. 정의에 의해 G-감축가능한 부분식은 함수형이 아니다. 그러나 표식 분류 형태 4인 부분식은 함수형이 아닌데에도 불구하고 내부에 컴비네이터 적용식을 포함하기 때문에 G-감축가능하지 않다. 다음 예 3.2에서 보는 바와 같이 형태 4인 부분식은 내부에 포함하고 있는 컴비네이터 적용식이 추출되어 추상화된다면 G-감축가능하게 된다.

[예 3.2] 그림 1은 주어진 컴비네이터식에 대하여 표식 형태를 표시하고 그림 2는 프로그램 변화에 의해 수정된 표식 형태를 나타낸다. 각 노드 □, ○, △, ◇ 는 각각 표식 형태 1, 2, 3, 4를 나타내고 이때 x'은 추상화과정에서 도입된 새로운 변수이다.

E = + (*3 x) ($ y)



(그림 1) 표식을 포함한 주어진 식 E
(Fig. 1) An annotated expression E



(그림 2) 변환된 G-감축가능한 식 E
(Fig. 2) A G-reducible expression E

이제 G-표현가능하지 않고 함수형이 아닌 형태 2의 부분식을 G-감축가능한 형태 2의 컴비네이터식으로 변환 하는 변환 함수 G를 정의한다.

[정의 3.4] 변환 함수 $\mathcal{G}$

변환 함수 G는 주어진 컴비네이터식 E에 대하여 $\mathcal{G}[E] = E[x_1/e_1, \ldots, x_n/e_n]$ 이고 $W = \{\langle x_1, e_1\rangle, \ldots, \langle x_n, e_n\rangle\}$ 임을 만족한다.

E는 표식을 포함하는 컴비네이터식이고 G[E]는 수

정된 표식을 갖는 결과식이다. 이때 W(where-절이라 한다)는 ⟨x, e⟩와 같은 튜플들의 집합이고 E에 대한 대치(substitution)을 나타낸다. x는 E에서 새로운 변수이고 e는 다음의 알고리즘에 의해 E로부터 추출된 부분식이다.

알고리즘:

주어진 입력식 E에 대하여 $W=\phi$라 놓는다. 이때 E가 다른 어떤 컴비네이터식의 부분식이 아닐 때 초기식(initial expression)이라 하자.

(i) E가 적용식이 아니면 $G[E]=E$이고 W는 변하지 않는다.

(ii) $E=MN$일 때 $M=M_1M_2...M_k$이고 M_1은 적용식이 아니라 하고 다음에서 E의 표식 형태에 따른다.

형태 1: $\mathcal{G}[E]=\mathcal{G}[M]\mathcal{G}[N]$.

형태 2: $\mathcal{G}[E]=E$.

형태 3: $\mathcal{G}[E]=\mathcal{G}[M]\mathcal{G}[N]$.

형태 4:

- M_1이 기본 연산자이면
- $\mathcal{G}[E]=\mathcal{G}[M]\mathcal{G}[N]$이고 E의 표식은 형태 2(G-감축가능)가 된다.
- M_1이 컴비네이터이고 E가 초기식이 아니면 $\mathcal{G}[E]=x$이고 E의 표식은 형태 2가 되고 튜플 $\langle x, \mathcal{G}[M]\mathcal{G}[N]\rangle$이 W에 포함된다.
- M_1이 컴비네이터이고 E가 초기식이면 $\mathcal{G}[E]=\mathcal{G}[M]\mathcal{G}[N]$가 된다.

튜플 $\langle x, \mathcal{G}[M]\mathcal{G}[N]\rangle$이 W에 포함되면 W는 $\mathcal{G}[M]$와 $\mathcal{G}[N]$에 의해 얻어지는 튜플들도 포함할 수 있다. 이것은 후에 해석과정에서 병렬성(parallelism)을 높히는 효과를 줄 것이다. 또한 형태 4의 경우에 초기식의 처리를 구분하는 이유는 다음과 같은 불필요한 변수의 등장을 피하기 위한 것이다.

$$\mathcal{G}[E]=x \text{ where } W=\{\langle x, \mathcal{G}[M]\mathcal{G}[N]\rangle\}$$

[정리 3.1] 주어진 컴비네이터식 E에 변환 함수 $\mathcal{G}$를 취한 결과식 $\mathcal{G}[E]$에서 형태 2의 표식을 갖는 부분식은 기본 연산자와 상수, 함수형이 아닌 변수로만 구성된다.

(증명) 정의 3.2와 3.3을 이용하여 컴비네이터식의 구문에 대한 귀납으로 증명될 수 있으며 특히 표식 형태 4인 컴비네이터식 $E=E_1E_2$에 대하여 E_1이 컴비네이터인 경우는 결과식이 새로운 변수가 되므로 쉽게 증명되고 이를 이용하여 E1이 기본 연산자인 경우를 증명할 수 있다.■

사실, 정리 3.1은 표식 형태 2인 컴비네이터식에서 내부의 변수들이 어떤 상수값으로 바인드되기만 하면 컴비네이터식 자체를 상수로 간주할 수 있도록 한다. 따라서 프로세스망 리덕션에서 G-감축가능한 식의 연산과정이 프로세스의 행위로 관찰되지 않고 은폐(hide)될 수 있다.

4. 컴비네이터 프로그램의 해석법

이제 앞절의 결과로 얻어진 표식을 포함하는 컴비네이터 프로그램을 프로세스망으로 표현하는 해석법에 대하여 설명한다. 여기서 설명하는 해석법은 Glauert의 람다 계산 해석법을 기반으로 하고 있지만 두가지 면에서 다르다; 첫째는 람다식을 위한 것이 아니라 컴비네이터식을 위한 해석법이다. 따라서 컴비네이터 정의에 대한 해석도 포함한다. 둘째는 컴비네이터를 포함하지 않는 부분식은 그래프 리덕션 함수를 이용하여 연산된다. 이것은 프로세스망 리덕션 모델이 다른 계산 모델과 결합될 수 있음을 보여준다.

4.1 컴비네이터 값호출 해석

G-감축가능한 컴비네이터식은 그래프 리덕션 함수를 포함하는 프로세스망으로 구성된다. 우리는 G-감축가능한 컴비네이터식에 대응되는 프로세스망을 초어(chore) 프로세스라고 부르며 이것은 내부에 컴비네이터 적용을 위한 프로세스를 포함하지 않는다.

컴비네이터식은 람다 추상형 대신에 컴비네이터 \$를 포함하므로 본 논문의 해석법의 대상은 다음의 구문을 갖는다.

$$E := x \mid K \mid MN \mid \$$$

이제 컴비네이터식 E에 대한 프로세스망 구성을 위해 값호출 해석법을 정의할 수 있다.

[정의 4.1] 컴비네이터식 값호출 해석

$[x]u = \text{rec } X.\ u?v.\ (x!v \mid X)$ (x는 변수)

$[K]u = \text{rec } X.\ u?v.\ (v!K \mid X) = Ch[K]u$ (K는 상수)

$[MN]u = Ch[MN]u$ (MN이 G-감축가능인 경우)
$[MN]u = (f!(a, u) \mid [M]f \mid [N]a)\backslash f \backslash a$ (MN이 G-감축가능이 아닌 경우)

Ch[_]u는 초어 프로세스의 해석을 나타낸다. G-감축가능한 적용식은 그래프 리덕션 함수 GR을 포함하도록 해석된다.

[정의 4.2] 초어 프로세스 해석
컴비네이터식 E가 G-감축가능하고 FN(E) $=\{x_1,\ldots,x_n\}$일 때

(i) E가 where-절 $W = \{\langle z_1, E_1\rangle,\ldots,\langle z_m, E_m\rangle\}$를 갖으면
$$Ch[E]u = u?q.\ (x1!k1 \mid\ldots\mid x_n!k_n \mid k_1?y_1.\ \ldots\ k_n?y_n.\ (q!GR(E[y_1/x_1, \ldots, y_n/x_n])) \mid [E_1]z_1 \mid \ldots \mid [E_m]z_m)\backslash k_1\backslash\ldots\backslash k_n$$

(ii) 그렇지 않으면,
$$Ch[E]u = u?q.\ (x_1!k_1 \mid \ldots \mid x_n!k_n \mid k_1?y_1.\ \ldots\ k_n?y_n.\ q!GR(E[y_1/x_1, \ldots, y_n/x_n])\backslash k_1\backslash\ldots\backslash k_n$$

그래프 리덕션 함수 GR은 E의 변수 x_i가 특정값 y_i로 바인드되기만 하면 델타 규칙에 의해 E의 정규형을 구할 수 있다. GR을 포함한 프로세스는 where-절의 컴비네이터식에 대한 프로세스들과 서로 병행되며 그래프 리덕션 과정은 프로세스망 리덕션내에서 은폐된다.

완전한 컴비네이터 프로그램을 대상으로 하기 위해서는 컴비네이터 정의에 대한 프로세스가 미리 정의될 수 있어야 한다. 다음 정의는 컴비네이터 정의에 대한 해석을 커리(curried) 함수형으로 나타낸다.

[정의 4.3] 컴비네이터 정의의 해석
n개의 인자를 갖는 컴비네이터 \$의 정의에 대하여, $(i \le n)$

$$[\$\ x_1 \ldots x_n = E,$$
$$[\$]u = u?(x_1, q).\ [\$\ x_1]q$$
$$[\$\ y_1 \ldots y_i]u = u?(x_i+1, q).\ [\$\ y_1 \ldots y_i, x_i+1]q$$
$$[\$\ y_1 \ldots y_n]u = [E[y_1/x_1, \ldots, y_n/x_n]]u$$

이 해석법은 람다 추상형에 대한 람다 계산 해석법과 유사하지만 컴비네이터 프로세스가 미리 정의된다는 이유로 프로세스의 재귀형태는 생략될 수 있다.

[예 4.1] 다음 컴비네이터 프로그램을 생각해 보자.

$$\$_1\ x = +\ x\ 1$$
$$\$_2\ x\ y = +\ 2\ (\$1\ (-x\ y))$$
$$\$prog = \$_2\ 5\ (-77\ 3)$$

컴비네이터 프로그램에 대한 표식과 변환 함수 $\mathcal{G}$에 의해서 다음의 수정된 컴비네이터식을 얻을 수 있다. 간략한 설명을 위해 W를 포함하는 컴비네이터식을 $\mathcal{G}[E] = E[x_1/e_1, \ldots, x_n/e_n]$ where $x_1 = e_1; \ldots; x_n = e_1;$와 같이 표현한다.

$$\mathcal{G}[+\ x\ 1] = +\ x\ 1$$
$$\mathcal{G}[+\ 2\ (\$_1\ (-x\ y))] = +\ 2\ x_1 \text{ where } x_1 = \$_1\ x_2; x_2 = -x\ y$$
$$\mathcal{G}[\$2\ 5\ (-77\ 3)] = \$_2\ 5\ x \text{ where } x = -77\ 3$$

이제 정의 4.1, 2, 3의 해석법을 이용하여 각 컴비네이터 정의와 컴비네이터식에 대한 프로세스를 다음과 같이 구성할 수 있다.

$$[\$_1]u = u?(x, q).\ q?r.\ (x!k \mid k?y.\ r!GR(+\ y\ 1))\backslash k$$
$$[\$_2]u = u?(x, q_1).\ q_1?(y, q_2).\ q_2?v.\ (x_1!k_1 \mid k_1?t_1.\ v!GR(+\ 2\ t_1) \mid f!(a, x_1) \mid [\$_1]f \mid [x_2]a \mid x_2?v.\ (x!k_2 \mid y!k_3 \mid k_2?t_2.\ k_3?t_3.\ v!GR(-t_2\ t_3)))\backslash k_1\backslash k_2\backslash k_3\backslash x_1\backslash x_2\backslash a\backslash f$$
$$[\$_2\ 5\ (-77\ 3)]u = ([\$_2]g \mid g!(a, f) \mid [5]a \mid f!(b, u) \mid b?q.\ q!GR(-77\ 3))\backslash g\backslash f\backslash a\backslash b$$

4.2 If-식의 값호출 해석

여기서는 지연 연산을 위한 해석법을 다루기에 앞서, 스트릭트한 기본 연산자와는 다른 IF 연산자에 대하여 앞에서 정의한 초어 프로세스 해석법이 어떤 효과를 가져오는 지를 설명한다. 초어 프로세스는 기본적으로 값호출 연산을 따른다. 즉, 그래프 리덕션 함수가 수행되기 위해서는 주어진 식내에 존재하는 모든 변수들이 먼저 바인드되어야 한다. 이것이 if-식에도 그대로 적용되어 컴비네이터식(7)과 같은 if-식은 변환 함수 G에 의해 식(8)과 같이 변환된다.

$$IF\ M_1\ M_2\ M_3 \quad (7)$$
$$IF\ \mathcal{G}[M_1]\ \mathcal{G}[M_2]\ \mathcal{G}[M_3] \quad (8)$$

이때 M_1, M_2, M_3가 모두 where-절을 생성한다면 변환식은 식(9)가 되고 초어 프로세스 해석에 의해 식(10)과 같은 프로세스망을 구성한다.

$$IF\ M_1'\ M_2'\ M_3' \text{ where } z_1 = E_1; \ldots; z_m = E_m \quad (9)$$
$$u?q.\ (\underline{x_1!k_1 \mid \ldots \mid x_n!k_n \mid k_1?y_1.\ \ldots\ k_n?y_n.} \quad (10)$$

$$q!GR((\text{IF } M_1' \; M_2' \; M_3')[y_1/x_1,\ldots,y_n/x_n]) \mid [E_1]z_1 \mid \ldots \mid [E_m]z_m)\backslash k_1\backslash\ldots\backslash k_n\backslash z_1\backslash\ldots\backslash z_m$$

이때 변수들의 관계는 $\{x_1,\ldots,x_n\} \supset \{z_1,\ldots,z_m\}$가 된다. 프로세스 식(10)에서 밑줄친 부분은 그래프 리덕션 함수가 IF 연산을 하기전에 if-식내의 모든 변수들을 바인드하기 위한 것이다. 변환과정에서 새로이 등장한 where-절 변수 $\{z_1,\ldots,z_m\}$뿐 아니라 식(7)에 있었을 변수들중 IF M_1' M_2' M_3'에 남아 있는 것까지도 모두 바인드된다. 식(7)에 나타나던 변수들중에는 추출되어 where-절로 추상화되어버린 것이 있을 수 있다는 것에 유의하자.

프로세스식(10)은 경우에 따라서 연산자 IF의 세 인자가 동시에 연산될 수 있음을 보여준다. 즉, 세개의 인자를 연산하는 모든 처리가 IF에 대한 그래프 리덕션전에 끝날 수 있다. 이것은 병렬성의 논의에서 speculative 병렬성이라고 부르는 형태이다. IF 연산자가 본질적으로 두번째와 세번째 인자에 대하여 넌스트릭트하기 때문에 이 두 인자의 연산이 모두 요구되지는 않지만 초어 프로세스는 값호출 연산에 의해 speculative 병렬성을 나타낼 수 있다. 다음은 값호출 연산에 의해 speculative 병렬성을 나타내는 프로세스망을 구성하는 구체적인 예를 보여준다.

[예 4.2] 다음 fibonacci 컴비네이터 프로그램은 if-식을 포함하는 전형적인 재귀함수이다.

$$\$_{fibo}\ x = \text{IF } (\text{or}(=x\ 0)\ (=x\ 1))\ 1\ (+\ (\$_{fibo}\ (-x\ 1))\ (\$_{fibo}\ (-x\ 2)))$$

컴비네이터식에 대한 표식과 변환에 의해 얻어진 결과는 다음과 같다.

$$\mathcal{G}[\text{IF } (\text{or } (=x\ 0)\ (=x\ 1))\ 1\ (+\ (\$_{fibo}\ (-x\ 1))\ (\$_{fibo}\ (-x\ 2)))]$$
$$= \text{IF } (\text{or } (=x\ 0)\ (=x\ 1))\ 1\ (+\ x_1\ x_2)$$
$$\text{where } x_1 = \$_{fibo}\ x_3;\ x_2 = \$_{fibo}\ x_4;\ x_3 = -x\ 1;\ x_4 = -x\ 2$$

여기서 if-식의 else 부분이 컴비네이터 적용식을 포함하므로 x_1과 x_2가 도입되고 이에 따른 where-절이 생성된다. 이제 정의 4.1, 2, 3의 해석법에 따라 다음과 같은 프로세스가 생성된다.

$$\begin{aligned}[\$_{fibo}]u = u(x, q).\ q?v.\ (&x!k \mid x_1!k_1x \mid x_2!k_2 \mid k?y.\ k_1?y_1.\ k_2?y_2.\\ &v!GR(\text{IF } (\text{or } (=y\ 0)\ (=y\ 1))\ 1\ (+\ y_1\ y_2))\\ &\mid ([\$_{fibo}]f \mid f!(a, x_1) \mid [x_3]a \mid x_3?v_1.\ (x!k_3 \mid k_3?y_3.\ v_1!GR\\ &(-y_3\ 1))\backslash k_3)\backslash f\backslash a\backslash x_3\\ &\mid ([\$_{fibo}]g \mid g!(b, x_2) \mid [x_4]b \mid x_4?v_2.\ (x!k_4 \mid k_4?y_4.\ v_2!GR\\ &(-y_4\ 2))\backslash k_4)\backslash g\backslash b\backslash x_4)\backslash k\backslash k_1\backslash k_2\backslash x_1\backslash x_2\end{aligned}$$

컴비네이터 $\$_{fibo}$에 대한 프로세스는 if-식에 대한 그래프 리덕션과 else 부분의 덧셈 연산을 위한 x_1, x_2의 연산이 동시에 이루어지도록 구성된다. 따라서 if-식의 조건부분이 true가 되는 경우에도 연산할 필요가 없는 ($\$_{fibo}$ -1)나 ($\$_{fibo}$ -2) 등이 연산을 위해 프로세스망을 확장하고 이것은 무한히 계속될 수 있다. 이러한 speculative 병렬성은 구현상의 문제들이 산재하고 재귀 함수의 값호출 해석을 불가능하게 하기 때문에 함수언어 구현에서 잘 받아들여지지 않고 있다. 따라서 일반적으로 많이 사용되는 factorial이나 fibonacci와 같은 전형적인 재귀 함수들도 4.1절의 해석법으로는 표현하기 어렵다. 이를 해결하는 방법으로 우리는 지연 연산을 이용하고자 한다.

4.3 컴비네이터 지연 해석

이제 초어 프로세스와 지연 연산이 어떻게 조합될 수 있는가에 대하여 설명한다. 컴비네이터식의 지연 연산을 위한 프로세스 해석은 정의 4.1의 값호출 해석으로부터 적용식에 대한 부분을 다음 Glauert의 해석으로 대체하면 간단히 얻을 수 있다.

$$[MN]u = (f!(t, u) \mid [M]f \mid \text{rec } X.\ t?v.\ (a!v \mid [N]a \mid X)\backslash a)\backslash f\backslash t \tag{11}$$

또한 4.1절에서 보여준 바와 같이 초어 프로세스를 이용하여 지연 해석에도 그래프 리덕션을 결합할 수 있다. 그러나 여기서 몇가지 언급해야 할 문제가 있다. 정의 4.1과 같은 방법으로 컴비네이터 적용식에 대하여 식(11)의 해석법을 취하고 G-감축가능한 적용식에 대하여는 초어 프로세스로 변환한다면 컴비네이터 적용식은 지연 연산을 나타낸다 해도 G-감축가능한 적용식의 지연 연산을 초어 프로세스가 보여줄 수 있느냐하는 것이다. 그럼에도 불구하고 초어 프로세스에서 이용하는 그래프 리덕션 함수는 값호출 연산을 연상하게 한다. 그 이유는 그래프 리덕션 함수 GR은 함수 연산이 시작되기 전에 주어진 식의 모든 변수들이 바인드되기 때문이다. 따라서 단순히 그래프 리덕션 함수를 지연 연산을 하도록 정의한다고 해도 값호출의 경우와 다를 것이 없다. 그러나 해결방

법 또한 여기에 있다. 그래프 리덕션 함수의 인자가 되는 G-감축가능한 부분식들은 기본 연산자에 의한 적용식이기 때문에 모든 기본 연산자가 자신의 인자에 스트릭트(strict)하다면 지연 연산이나 값호출 연산은 종료 조건(termination condition)이나 시멘틱스에 있어서 동일한 결과를 갖는다. 따라서 그래프 리덕션 함수 GR을 이용하는 것이 지연 연산이나 값호출 연산이냐하는 문제에서 벗어날 수 있다. 이것은 스택 계산 함수와 같이 지연 연산을 표현 하기 어려운 계산 모델의 적용이 가능한 이유도 된다.

그러나 4.2절에서와 같이 기본 연산자는 인자에 대하여 넌스트릭트한 IF를 포함한다. 기본연산자 IF는 세개의 인자중에서 두번째와 세번째인자에 대하여 넌스트릭트하므로 4.1절의 해석법으로는 지연 연산을 표현할 수 없다. 따라서 if-식을 위한 초어 프로세스는 지연 연산을 나타내기 위해 수정되어야 하고 그 전에 변환 함수 $\mathcal{G}$의 수정도 불가피하다.

[정의 4.4] 지연 연산을 위한 If-식의 변환

If-식 E = IF M_1 M_2 M_3에 대하여, (x_1과 x_2는 새로운 변수)

$\mathcal{G}$[E] = IF $\mathcal{G}$[M_1] x_1 x_2 where $x_1 = \mathcal{G}_1[M_2]; x_2 = \mathcal{G}_2[M_3]$ 이고

$\mathcal{G}_1[M_2]$와 $\mathcal{G}_2[M_3]$는 각각 자신의 where-절 W_1과 W_2를 갖는다.

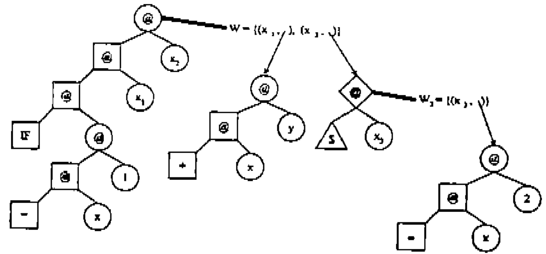
변환 함수 $\mathcal{G}$에 대한 첨자는 각각 대응되는 새로운 where-절을 생성한다는 것을 의미한다. 따라서 $\mathcal{G}_1[M_2]$와 $\mathcal{G}_2[M_3]$르부터 생성되는 새로운 where-절 W_1과 W_2는 $\mathcal{G}$[E]의 where-절 W에 내포될 수 있다. 예를 들어 다음 if-식(12)는 where-절이 내포된 식(13)과 같이 변환되고 이를 그래프 표현으로 나타내면 그림 3이 된다.

IF(= x 1) (+ x y) ($(− x 2))   (12)

IF(= x 1) x1 x2 where x1 = + x y;   (13)
(x2 = $ x3 where x3 = − x 2)

이제 변환된 G-감축가능한 if-식에 대하여 지연 연산을 나타내는 초어 프로세스를 정의한다. IF 연산자를 위한 초어 프로세스는 IF의 그래프 리덕션 결과를 링크 명칭의 역할을 하는 변수가 되도록 한다.

[정의 4.5] If-식의 지연 해석



(그림 3) where-절의 내포를 보여주는 if-식의 그래프 표현
(Fig. 3) A graph representation for an if-expression with nested where-clause

다음의 G-감축가능한 if-식에 대하여($1 \le i, j \le n, i \ne j$)

$E = IF\ M_1\ z_i\ z_j$ where $z_1 = E_1; \ldots; z_n = E_n$

$Ch[E]u = u?q.\ (x_1!k_1 \mid \ldots \mid x_m!k_m \mid k_1?y_1. \ldots k_m?y_m.$
$\quad w!GR(E[y_1/x_1, \ldots, y_m/x_m])$
$\quad \mid w?z.\ z!q \mid [E_1]z_1 \mid \ldots \mid [E_n]z_n) \backslash k_1 \backslash \ldots \backslash k_m \backslash w \backslash z_1$
$\quad \backslash \ldots \backslash z_n$

$(\{x_1, \ldots, x_m\} = FN(M_1))$

여기서 G-감축가능한 if-식의 z_i와 z_j는 정의 4.4에 의해 항상 변수를 나타낸다. 또한 M_1의 자유변수들만을 그래프 리덕션전에 요구하는 $x_1!k_1 \mid \ldots \mid x_m!k_m$ 부분은 초어 프로세스가 지연 연산을 나타내고 있음을 보여준다. 이로써 M_2와 M_3에 대한 각각 독립적인 프로세스가 생성되고 그중 하나만 수행됨으로써 speculative 병렬성은 제거되고 M_1에 대한 프로세스가 끝난 후에 비로서 M_2나 M_3를 요구하는 지연 연산을 가능하게 한다.

정의 3.4와 4.1에 각각 정의 4.5, 6을 포함시키면 다음 정의로부터 주어진 컴비네이터식의 지연 연산을 나타내는 프로세스망을 구성할 수 있다.

[정의 4.6] 컴비네이터식 지연 해석

$[x]u = rec\ X.\ u?v.\ (x!v \mid X)$
$[K]u = rec\ X.\ u?v.\ (v!K \mid X) = Ch[K]u$
$[MN]u = Ch[MN]u$ (MN이 G-감축가능인 경우)
$[MN]u = (f!(t, u) \mid [M]f \mid rec\ X.\ t?v.\ (a!v \mid [N]a \mid X) \backslash a)$
$\quad \backslash f \backslash t$ (G-감축가능이 아닌 경우)

완전한 컴비네이터 프로그램에 대한 실예를 다음에서 볼 수 있다. 컴비네이터 정의에 대한 해석은 정의 4.3을 그대로 따르고 있다.

[예 4.4] 아래와 같은 컴비네이터 프로그램이 주어진

다고 하자.

$$\$_1\ x\ y = \mathrm{IF}\ (\langle\rangle\ x\ 0)\ y\ (+y\ 1)$$
$$\$_{prog} = \$_1\ (+2\ 3)\ (+3\ 4)$$

변환함수 G에 의해 if-식의 then 부분과 else 부분은 where-절로 추출된다.

$$\mathcal{G}[\mathrm{IF}\ (\langle\rangle\ x\ 0)\ y\ (+y\ 1)] = \mathrm{IF}\ (\langle\rangle\ x\ 0)\ x_1\ x_2$$
$$\text{where } x_1 = y; x_2 = +y\ 1$$
$$\mathcal{G}[\$_1\ (+2\ 3)\ (+3\ 4)] = \$_1\ x_3\ x_4 \text{ where } x_3 = +\ 2\ 3;\quad x_4 = +3\ 4$$

이제 지연 해석을 통해 다음의 프로세스를 얻을 수 있다.

$$[\$_1]u = u?(x,\ q).\ q?(y,\ v).\ v?r.\ (x!k \mid k?y_1.\ w!GR\ (\mathrm{IF}\ (\langle\rangle\ y_1\ 0)\ x_1\ x_2) \mid w?v.\ r!v \mid [y]x_1 \mid [+y\ 1]x_2) \backslash k \backslash w$$
$$[\$_{prog}]u = ((([\$_1]f \mid f!(t_1,\ g) \mid \mathrm{rec}\ X.\ t_1?v.\ (a!v \mid [x_3]a) \backslash a) \backslash f \backslash t_1 \mid g!(t_2,\ u) \mid \mathrm{rec}\ X.\ t_2?v.\ (b!v \mid [x_4]b) \backslash b) \backslash g \backslash t_2 \mid [+2\ 3]\ x_3 \mid [+3\ 4]x_4) \backslash x_3 \backslash x_4$$

위와 같이 구성된 프로세스망에 대한 리덕션은 결국 두개의 프로세스 $[y]x_1$과 $[+y\ 1]x_2$중에서 하나만을 호출하게 되고 여기서는 $[y]x_1$만이 연산된다.

4.4 그래프 리덕션의 공유

이제 그래프 리덕션 결과의 공유를 위한 함수 GR을 포함하는 출력 프로세스의 재귀 호출을 설명한다. Glauert의 기본연산자를 위한 Const 프로세스는 2장의 식(1)과 같이 상수에 대한 재귀 출력을 표현하고 있으며 컴비네이터 해석으로 생성되는 프로세스도 그래프 리덕션 결과에 대한 재귀 호출이 요구된다. 이것은 상수나 그래프 리덕션의 결과가 프로그램의 서로 다른 부분식에 의해 공유될 수 있도록 한다.

[정의 4.7] 그래프 리덕션의 공유를 위한 초어 프로세스

그래프 리덕션의 결과가 재귀적으로 출력되기 위하여 초어 프로세스에 대한 정의 4.2와 4.5의 해석은 다음과 같이 수정된다.

(i) G-감축가능한 식 E가 where-절에 $z_1 = E_1; \ldots; z_m = E_m$을 갖는다면 E의 초어 프로세스는 다음과 같이 생성된다.

$$Ch[E]u = u?q.\ (x_1!k_1 \mid \ldots \mid x_n!k_n \mid k_1?y_1.\ \ldots\ k_n?y_n.\ \ldots\ w!GR(E[y_1/x_1,\ \ldots,y_n/x_n]) \mid \underline{w?v.\ (q!v \mid \mid \mathrm{rec}\ X.\ u?q.\ (q!v \mid X))} \mid [E_1]z_1 \mid \ldots \mid [E_m]z_m) \backslash k_1 \backslash \ldots \backslash k_n \backslash w \backslash z_1 \backslash \ldots \backslash z_m$$
$$(FN(E) = \{x_1, \ldots, x_n\})$$

(ii) G-감축가능한 if-식 $E = \mathrm{IF}\ M1\ z_i\ z_j$ where $z_1 = E_1; \ldots; z_n = E_n$ $(1 \le i,\ j \le n,\ i \ne j)$에 대하여 다음의 지연 연산을 위한 초어 프로세스가 생성된다.

$$Ch[E]u = u?q.\ (x_1!k_1 \mid \ldots \mid x_m!k_m \mid k_1?y_1.\ \ldots k_m?y_m.\ (w!GR(E[y_1/x_1,\ldots,y_m/x_m]) \mid \underline{w?z.\ (z!q \mid \mathrm{rec}\ X.\ u?q.\ (z!q \mid X))} \mid [E_1]z_1 \mid \ldots \mid [E_n]z_n) \backslash k_1 \backslash \ldots \backslash k_m \backslash w \backslash z_1 \backslash \ldots \backslash z_n$$
$$(\{x_1, \ldots, x_m\} = FN(M_1))$$

위에서 그래프 리덕션의 결과는 밑줄친 재귀 프로세스로 전달되고 이 재귀 프로세스는 전달 받은 값을 반복적으로 출력한다. 이때 (i)의 경우는 정의 4.1에서 상수에 대한 프로세스와 같은 형태가 되고 if-식인 (ii)의 경우는 그래프 리덕션의 결과가 새로운 변수로 생성된 링크 명칭이 되므로 정의 4.1에서 변수 x에 대한 프로세스와 같은 형태가 된다. 결과적으로 그래프 리덕션에 의해서 얻어진 결과는 반복적으로 다른 병렬 프로세스에 전달될 수 있으므로 초어 프로세스가 반복적으로 호출되는 경우에도 그래프 리덕션 과정은 반복되지 않는다.

[예 4.5] 다음 컴비네이터 정의는 전형적인 재귀 함수 factorial을 나타낸다. 재귀 함수로부터 생성된 프로세스가 내부의 초어 프로세스를 반복적으로 호출할 때 그래프 리덕션 결과는 공유된다.

$$\$_{fac}\ x = \mathrm{IF}\ (=x\ 0)\ 1\ (*\ x\ (\$_{fac}\ (-x\ 1)))$$

factorial 컴비네이터의 지연 연산을 위한 변환은 다음의 결과를 보여준다.

$$\mathcal{G}[\mathrm{IF}\ (=x\ 0)\ 1\ (*\ x\ (\$_{fac}\ (-x\ 1)))]$$
$$= \mathrm{IF}\ (=x\ 0)\ x_1\ x_2 \text{ where } x_1 = 1; (x_2 = *x\ x_3 \text{ where } x_3 = \$_{fac}\ x_4; x_4 = -x\ 1)$$

변환된 컴비네이터식이 where-절을 내포된 형태로 갖는다는 것에 유의하면서 지연 해석을 통해 얻어지는 프로세스는 다음과 같다.

$$[\$_{fac}]u$$
$$= u?(x,\ q).\ q?v.\ (x!k \mid k?y.\ w!GR(\mathrm{IF}\ (=y\ 0)$$

$$x_1\ x_2) \mid w?z.\ (z!v \mid rec\ X.\ u?v.\ (z!v \mid X))$$

$$\mid [1]x_1 \mid [*x\ x_3]x_2 \mid [\$_{fac}\ x_4]x_3 \mid [-x\ 1]x_4) \backslash k \backslash w \backslash x_1 \backslash x_2 \backslash x_3 \backslash x_4$$

$$= u?(x,\ q).\ q?v.\ (x!k \mid k?y.\ \underline{w!GR(IF\ (=y\ 0)\ x_1\ x_2) \mid w?z.}$$

$$\underline{(z!v \mid rec\ X.\ u?v.\ (z!v \mid X))}$$

$$\mid\ rec\ X.\ x_1?v.\ (v!1 \mid X) \mid x2?r.\ (x!k_1 \mid x_3!k_2$$

$$\mid k_1?y_1.\ k_2?y_2.\ w_1!GR(*\ y_1\ y_2) \mid w_1?v.\ (r!v \mid rec\ X.\ x_2?r.$$

$$(r!v \mid X))$$

$$\mid ([\$_{fac}]f \mid\ f!(t,\ x_3) \mid\ rec\ X.\ t?v.\ (a!v \mid rec\ Y.\ a?r.$$

$$(x_4!r \mid Y) \mid X) \backslash a) \backslash t \backslash f \mid x_4?r.(x!k_3$$

$$\mid k_3?y_3.\ \underline{w_2!GR(-y_3\ 1) \mid w_2?v.\ (r!v \mid rec\ X.\ x_4?r.}$$

$$\underline{(r!v \mid X)))} \backslash k_3 \backslash w_2) \backslash k_1 \backslash k_2 \backslash w_1) \backslash k \backslash w \backslash x_1 \backslash x_2 \backslash x_3 \backslash x_4$$

컴비네이터 $\$_{fac}$에 대한 프로세스는 밑줄친 세 부분에 그래프 리덕션의 공유를 위한 재귀 프로세스를 포함한다.

5. 분석및 평가

이제 초어 프로세스를 도입함으로써 프로세스망 리덕션에서 구체적으로 발생하는 효과를 설명한다. 초어 프로세스는 프로세스망 리덕션 과정중에 암시적으로 즉, 외부적으로 관찰됨이 없이 그래프 리덕션이 수행된다. 따라서 그래프 리덕션으로 연산되는 부분식은 따로 프로세스망 리덕션 단계를 나타내지 않으므로 전체 프로세스들의 통신 행위는 그만큼 감소된다. 이것은 프로세스 모델의 프로세스가 계산 능력(computability)을 갖음으로써 계산 단위를 증가시키는 역할을 한다. 우리는 이러한 효과를 보이기 위해 B.Victor[21]의 MWB(Mobility Workbench) 버젼 0.111에서 각 해석법에 의해 생성된 프로세스망의 통신전이 회수를 구하였다.

다음의 컴비네이터는 각각 계산 부분을 갖지 않는 컴비네이터 I, 인자의 공유를 갖는 컴비네이터 $\$_{sh}$, 기본연산의 복합식을 갖는 $\$_{comp}$, IF 연산자를 포함하는 $\$_{abs}$, 재귀함수 fibonacci에 대한 컴비네이터 $\$_{fibo}$이다.

$$I\ x = x$$

$$\$_{sh}\ x = *x\ x$$

$$\$_{comp}\ x\ y = +\ (*\ x\ y)\ y$$

$$\$_{abs}\ x = IF\ (x\ \rangle = 0)\ x\ \ -x$$

$$\$_{fibo}\ x = IF\ (or\ (=x\ 0)\ (=x\ 1))\ 1\ (+(\$_{fibo}\ (-x\ 1))$$

$$(\$_{fibo}\ (-x\ 2)))$$

이들중에서 컴비네이터 I와 $\$_{sh}$, $\$_{comp}$는 초어를 갖는 값호출 해석과 지연 해석에 의해 식(14),(15), (16)과 같이 동일한 프로세스가 생성되며 컴비네이터 $\$_{abs}$는 지연 해석에 의해 초어를 갖지 않는 경우에 식(17), 초어를 갖는 경우에 식(18)의 프로세스를 생성한다.

$$[I]\,u = u?(x,\ q).\ rec\ X.\ q?v.\ (x!v \mid X) \qquad (14)$$

$$[\$_{sh}]\,u = u?(x,\ q).\ q?r.\ (x!k \mid k?y.\ w!GR(*y\ y) \mid w?v.$$

$$(r!v \mid rec\ X.\ q?r.(r!v \mid X))) \backslash k \backslash w \qquad (15)$$

$$[\$_{comp}]\,u = u?(x,\ q).\ q?(y,\ r).\ r?v.\ (x!k_1 \mid y!k_2 \mid k_1?m_1.$$

$$k_2?m_2.\ w!GR(+(*m_1\ m_2)\ m_2) \qquad (16)$$

$$\mid w?z.\ (v!z \mid rec\ X.\ r?v.\ (v!z \mid X))) \backslash k_1 \backslash k_2 \backslash w$$

$$[\$_{abs}]\,u = u?(x,\ q).\ (rec\ X.\ a_1?v.\ (x!v \mid X) \mid rec \qquad (17)$$

$$X.\ b_1?v.\ (v!0 \mid X)$$

$$\mid (a_1!k_1 \mid b_1!k_2 \mid k_1?m_1.\ k_2?n_1.\ Const(a,\ \rangle = m_1\ n_1))$$

$$\backslash k_1 \backslash k_2) \backslash a_1 \backslash b_1$$

$$\mid rec\ X.\ b?v.\ (x!v \mid X) \mid (rec\ X.\ a2?v.\ (x!v \mid X)$$

$$\mid (a_2!k_3 \mid k_3?m_2.\ Const(c,\ -m_2)) \backslash k_3) \backslash a_2 \mid a!k \mid k?v$$

$$Var(q,\ (IF\ v\ b\ c))) \backslash a \backslash b \backslash c \backslash k$$

$$[\$_{abs}]\,u = u?(x,\ q).\ q?r.\ (x!k \mid k?v.\ w!GR(IF(\ \rangle = v\ 0)$$

$$x_1\ x_2) \qquad (18)$$

$$\mid w?z.\ (z!r \mid rec\ X.\ q?r.\ (z!r \mid X)) \mid rec\ X.\ x_1?v.$$

$$(x!v \mid X)\ \mid x2?r.\ (x!k_1 \mid k_1?v.\ w_1!GR(-v) \mid w_1?z.$$

$$(r!z \mid rec\ X.\ x_2?r.\ (z!r \mid X)) \backslash k_1 \backslash w_1) \backslash k \backslash w$$

이때 컴비네이터 $\$_{abs}$에 대해서는 IF를 포함하므로 지연 해석만을 고려 하였다. 이것은 4. 2절에서 언급하였듯이 값호출 IF는 재귀 함수에서 이용할 수 없기 때문에 여기서는 이를 배제하였다. 또한 Glauert는 자신의 해석법에서 연산자 IF에 대한 프로세스를 정의하지 않았다. 그러나 여기서는 Glauert의 Const 프로세스와 대비되는 형태를 이용하여 식(19)와 같이 지연 IF에 대한 프로세스를 정의하고 이를 통해 본 논문의 방법과 비교하였다.

$$[IF\ P\ M\ N]\,u = ([P]a \mid [M]b \mid [N]c \mid a!k \mid k?v.$$

$$Var(u,\ (IF\ v\ b\ c))) \backslash a \backslash b \backslash c \backslash k \qquad (19)$$

$$\text{where } Var(u,\ z) = u?r.\ (z!r \mid Var(u,z))$$

표 2는 이들 컴비네이터들에 대한 프로그램이 프로세스망 리덕션에서 발생하는 통신 행위의 수를 나타낸다. 컴비네이터 I에 대한 프로그램은 그래프 리덕션을 포함하지 않는 프로세스로 해석되므로 초어를

갖는 경우와 그렇지 않은 경우에 통신 행위의 회수는 변하지 않는다. 그러나 컴비네이터 $\$_{sh}$와 $\$_{comp}$는 초어를 갖는 경우에 값호출이나 지연 해석으로 동일한 프로세스를 생성하지만 인자에 적용되는 방식의 차이 때문에 그래프 리덕션 함수를 갖는 초어 프로세스를 포함할 때 그래프 리덕션이 발생하면 할수록 더 적은 통신 행위를 기록한다. 또한, 컴비네이터 $\$_{abs}$에 대한 프로그램과 컴비네이터 $\$_{comp}$와 $\$_{abs}$가 함께 나타나는 프로그램도 마찬가지로 초어 프로세스를 갖는 경우에 통신 행위의 감소를 나타낸다.

〈표 2〉 프로세스망 리덕션에서 통신 행위의 회수
〈Table 2〉 The number of communication actions in process-net reduction

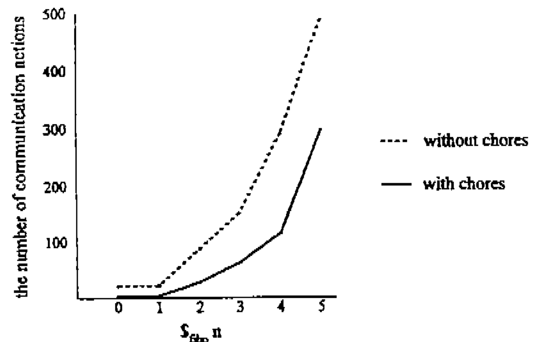
프로그램	초어를 갖지 않는 값호출 해석	초어를 갖지 않는 지연 해석	초어를 갖는 값호출 해석	초어를 갖는 지연 해석
199	4	5	4	5
$\$_{sh}$(* 4 5)	13	19	5	7
$\$_{comp}$ 3 6	15	18	9	11
$\$_{abs}$ −8	−	16	−	12
$\$_{comp}$ 5 ($\$_{abs}$ −3)	−	31	−	20

또한 재귀 함수에 대하여 각 프로세스 해석에 따른 비교 분석을 위해 컴비네이터 $\$_{fibo}$에 대한 결과를 표 3과 같이 얻을 수 있고 그림 4는 그 증가 추세를 보여준다. 이때 컴비네이터 $\$_{fibo}$에 대한 값호출 프로세스는 speculative 병렬성을 나타냄으로써 프로세스가 무한히 확장되어 결과를 얻을 수 없다. 따라서 지연 연산에 대해서만 초어 프로세스를 가지 않는 경우와 초어 프로세스를 갖는 경우로 나누어 비교하였다. 초어를 갖는 지연 해석에 의한 프로세스는 식(20)와 같다.

$$[\$_{fibo}]u = u?(x, q).\ q?v.\ (x!k \mid k?y.\ w!GR(IF\ (or\ (=y\ 0)\ (=y\ 1))\ x_1\ x_2) \tag{20}$$
$$\mid w?z.(z!v \mid rec\ X.\ q?v.\ (z!v \mid X)) \mid [1]x_1 \mid x_2?v.(x_3!k_1 \mid x_4!k_2 \mid k_1?y_1.k_2?y_2.w_1!GR(+\ y_1\ y_2)$$
$$\mid w_1?r.(v!r \mid rec\ X.\ x_2?v.\ (v!r \mid X)) \mid ([\$_{fibo}]f \mid f!(t_1, x_3) \mid rec\ X.\ t_1?v.\ (a!v \mid [x_5]a \mid X)$$
$$\mid x_5?v.(x!k_3 \mid k_3?y_3.\ w_2!GR(-y_3\ 1) \mid w_2?r.(v!r \mid rec\ X.\ x_5?v.(v!r \mid X)))\backslash k_3\backslash w_2)\backslash f\backslash a\backslash t_1\backslash x_5$$
$$\mid ([\$_{fibo}]g \mid g!(t_2, x_4) \mid rec\ X.\ t_2?v.\ (b!v \mid [x_6]b \mid X) \mid x_6?v.(x!k_4 \mid k_4?y_4.w3!GR(-y_4\ 2)$$
$$\mid w_3?r.(v!r \mid rec\ X.\ x_6?v.(v!r \mid X)))\backslash k_4\backslash w_3)\backslash g\backslash b\backslash t_2\backslash x_6)\backslash k_1\backslash k_2\backslash w_1\backslash x_3\backslash x_4)\backslash k\backslash w\backslash x_1\backslash x_2$$

〈표 3〉 재귀함수의 프로세스망에서 통신 행위의 회수
〈Table 3〉 The number of communication actions in process networks for a recursive function

S_{fibo}	초어를 갖지않는 지연 해석	초어를 갖는 지연 해석
0	22	8
1	22	8
2	88	35
3	155	62
4	289	116
5	490	197



(그림 4) 재귀함수의 통신 행위 증가 추세
(Fig. 4) The incremental trends of communication actions for a recursive function

이로써 우리는 초어 프로세스 해석법을 이용하면 프로세스 모델의 기본(atomic) 프로세스인 입력과 출력 프로세스를 확장하여 계산 능력을 갖는 계산(computable) 프로세스를 생성할 수 있다는 것을 알 수 있다. 또한 이러한 계산 프로세스는 프로세스의 계산 단위를 확장하고 따라서 외부로 관찰되는 프로세스망에서의 전이 단계가 그만큼 감소된다.

6. 결 론

본 논문은 병렬구문을 갖지 않는 컴비네이터 프로그램을 위한 값호출 해석법과 지연 해석법을 각각 제안하였다. 컴비네이터 해석법은 생성된 프로세스에 그래프 리덕션 함수를 포함시킴으로써 프로세스의 계산 단위를 확장하였을 뿐 아니라 프로세스 모델에 다른 계산 모델을 결합시키는 방법을 제시하였다. 새로운 해석법은 변환 함수의 도입으로 컴파일 시간의 오버헤드를 야기시키지만 실행 시간의 효율성을 얻을 수 있다.

본 논문의 결과에 의하면 함수 프로그램의 계산 모델인 리덕션 메카니즘이 프로세스망 리덕션과 그래프 리덕션의 결합된 형태로 구성될 수 있으며 이때의 그래프 리덕션은 프로세스망 리덕션 과정에서 은폐된다. 또한 그래프 리덕션 함수는 연산하는 부분식이 델타 규칙만으로도 가능한 상수식이 되므로 어떤 다른 그래프 리덕션 함수보다도 간단한 함수가 된다. 따라서 그래프 리덕션이 아닌 다른 간단한 계산 함수의 채용도 가능하고 어떤 계산 함수를 이용하더라도 함수의 수행은 지역적으로 이루어 질 수 있다. 계산 함수의 지역적인 수행은 그것이 기본 프로세스 즉, 출력 프로세스에 은폐되기 때문이다. 특히 초어 프로세스의 공유는 함수 프로그램의 성질중에서 부분식의 공유문제를 정확하게 재현한다.

본 논문은 컴비네이터 해석법에 대하여 설명하였지만 프로세스들의 동치 관계인 bisimulation을 규명하는 문제가 남아 있다. 그러나 이것은 Milner[15]의 방법으로 해결될 수 있을 것이다. 문제는 계산 함수로부터 얻어지는 시멘틱값을 프로세스의 동작 시멘틱스에 적용하는 방법이 필요하다는 것이다. 본 논문에서 이용한 계산 함수 GR은 부분 적용식을 결과로 내는 경우를 배제한다. 따라서 계산 함수를 좀더 복잡한 고계함수로 확장할 경우에 프로세스의 해석법 뿐아니라 변환 함수의 정의도 달라져야 할 것이다.

끝으로 본 논문에서 제시한 해석법은 함수 프로그램의 프로세스망 구현을 위한 새로운 컴파일 방법으로 전역 기억장소를 갖지 않는 병렬 환경에서의 함수 언어 구현에 적용될 수 있으며 CHAM[2]은 프로세스 모델의 시멘틱스를 규명하는 도구이지만 본 논문의 모델을 적용하기에 적절한 시작점이 될 수 있을 것이다.

참 고 문 헌

[1] S.Abramsky, "The lazy lambda calculus", Reseach topics in functional programming, ed. D.Turner, pp. 65-116, Addison Wesley, 1988.

[2] G.Berry and G. Boudol, "The Chemical Abstract machine", Proc. ACM Principles of Programming Languages, pp. 81-94, 1990.

[3] A.Giacalne, P.Mishra, and S. Prasad, "A symmetric integration of concurrent and functional programming", International Journal of Parallel Programming, vol. 18, no. 2, 1990.

[4] J.R.W.Glauert, L.Leth, and B. Thomsen, "A new translation of functions as processes", SemaGraph '91 Symposium, 1991.

[5] J.R.W.Glauert, "Asynchronous mobile processes and graph rewriting", Proc. PARLE, LNCS #605, Springer-Verlag, pp. 63-78, 1992.

[6] B.F.Goldberg, Multiprocessor execution of functional programs, Research Report YALEU/DCS/RR-618, Yale University, 1988.

[7] K.Honda and M. Tokoro, "An object calculus for asynchronous communication", Proc. ECOOP '91, LNCS #512, Springer-Verlag, pp. 133-147, 1991.

[8] P.Hudak and B.F.Goldberg, "Serial combinators: optimal grains of parallelism", LNCS #201, Springer-Verlag, pp. 382-388, 1985.

[9] R.J.M. Hughes, "Super-combinators: A New Implementation Method for Applicative Languages", Sym. on Lisp and Functional Prog., pp. 1-10, 1982.

[10] J.R.Kennaway and M.R.Sleep, "Expressions as processes", Proc. Lisp and Functional Programming, pp. 21-28, 1982.

[11] L.Leth, "Functional programs as reconfigurable networks of communicating processes", Ph.D. Thesis, Imperial College, London University, 1991.

[12] R.Milner, Calculus of communicating systems, LNCS #92, Springer-Verlag, 1980.

[13] R.Milner, Communication and concurrency, Prentice Hall, 1989.

[14] R.Milner, J.Parrow, and D.Walker, A calculus of mobile processes, Parts I and II, TR ECS-LFCS-89-85,86, Edinburgh University, 1989.

[15] R.Milner, "Functions as processes", Automata, languages and Programming, LNCS #443, Springer-Verlag, pp. 167-180, 1990.

[16] S.L.Peyton Jones, The implementation of functional programming languages, Prentice Hall, 1987.

[17] S.L.Peyton Jones, Parallel implementations of functional programming languages, The computer journal, vol. 32, no. 2, pp. 175-186, 1989.

[18] S.Prasad, Towards a symmetric integration of concurrent and functional programming, Ph.D. Thesis, State University of New York at Stony Brook, 1991

[19] J.H.Reppy, "CML: A higher-order concurrent language", Proc.ACM SIGPLAN '91, Conference on Programming Language Design and Implementation, pp. 293-305, 1991.

[20] B.Thomsen, Calculi for higher order communicating systems, Ph.D. Thesis, Imperial College, London University, 1990.

[21] B.Victor, A verification tool for the polyadic π-calculus. Licentiate thesis, Uppsala University, 1994

신 승 철

1987년 인하대학교 전자계산학과 졸업(이학사)
1989년 인하대학교 대학원 전자계산학과 졸업(이학석사)
1996년 인하대학교 대학원 전자계산학과 졸업(공학박사)
1994년~1996년 인하대학교 전자계산공학과 전임대우강사
1996년~현재 동양대학교 컴퓨터공학부 전임강사
1996년~현재 동양대학교 컴퓨터 정보센터 소장
관심분야: Language Semantics, Computational Model, Functional Languages, Concurrency Theory, Parallel Processing

유 원 희

1975년 서울대학교 공과대학 응용수학과 졸업(이학사)
1978년 서울대학교 대학원 계산학 전공(이학석사)
1985년 서울대학교 대학원 계산학 전공(이학박사)
1979년~현재 인하대학교 공과대학 전자계산공학과 교수
관심분야: 프로그래밍 언어(실시간 프로그래밍 언어, 함수 언어).