JBURG를 이용한 JIT컴파일러 생성에 관한 연구

강 경 우†

요 약

본 연구에서는 JBURG를 이용하여 JIT 컴파일러를 생성하는 방법을 제안하였다. JBURG는 Java를 위한 상향식 트리패턴 일치 코드생성기를 생성하는 도구이다. 본 연구에서 제안한 방법은 트리패턴 사이에 관계를 조사함으로 고안되었다. 제안된 방법은 패턴들 사이의 관계를 이용해 분석하고 불필요한 점검을 피할 수 있다는 점에서 기존의 연구결과인 JBURG보다 효율적이다. 필요한 분석들은 코드생성기를 만드는 시간에 수행되기 때문에 실제 코드를 생성해야 하는 컴파일 시간에 효율을 높일 수 있다.

A study on the generation of JIT compiler using JBURG

Kyung-Woo Kang†

ABSTRACT

In this paper, we propose a method of generating a JIT compiler using JBURG. JBURG is a tool of generating the code generator using bottom-up tree pattern matching for Java. Our method can be derived from some relations over tree patterns. The proposed scheme is more efficient than JBURG because we can avoid unfruitful tests with the smaller size of the scheme. Furthermore, the relevant analyses needed for this proposal are largely achieved at non compile time, which secures actual efficiency at compilation time.

키워드 : 컴파일러(Compiler), 코드생성기(Code Generator), 상향식 트리패턴 매칭(Bottom-Up Tree Pattern Matching)

1. 서 론

인터넷이 보편화됨에 따라 프로그램 코드도 "write-once run-everywhere"라는 특징을 요구하고 있다. 프로그램은 중간코드 형태로 저장되고 수요자의 요구에 따라 전송되며 수요자 컴퓨팅 환경에서 수행된다. 지금까지는 전송된 코드는 수요자의 컴퓨터에서 인터프리터를 이용해서 수행되었는데 컴파일러 기술의 발달로 점차 중간코드를 기계어로 변환 후 수행하는 JIT 컴파일러를 사용하는 추세이다. 이때 중요한 것은 JIT 컴파일러의 컴파일 속도이다[17].

많은 코드생성기들은 트리 구조로 표현된 원시 프로그램의 중간코드를 입력으로 받아서 목적프로그램을 생성한다. 트리 구조는 원시프로그램의 구문적, 의미적 구조를 잘 표현해 줄 수 있고 트리를 다루는 기술들이 단순하며 효율적이기 때문에 컴파일러의 중간표현으로 적당하다[14]. Hoffman과 O'Donnell에 의해 트리패턴 일치(tree pattern matching)가 정형화된 이후 동적 프로그래밍(dynamic programming)을 이용하는 트리패턴 일치는 코드생성기의 실제적인 기술로서 많이 사용되었다[1, 2, 4, 6, 7, 9, 11-13, 14, 16].

프로세서에서 제공하는 명령어와 중간코드 사이의 관계인 기계묘사를 입력으로 받아서 코드생성기를 만드는 도구는 자동으로 트리패턴 일치를 이용하는 코드생성기를 만들 수 있다. 프로세서들이 다양해짐에 따라 신속하게 개발해야 하는 코드생성기의 자동 생성의 요구는 점점 커지고 있다[2]. 자동화 과정에서 생성된 코드 생성기는 아직까지 효율적이지 못한 면이 많았는데 자동화에 관한 최근의 연구들은 코드생성기를 만드는 도구를 실용적으로 만들었다. 게다가 자동화는 어떤 아키텍처가 목적 코드의 크기와 실행시간 등에 미치는 영향을 쉽게 알아볼 수 있는 도구로써도 역할을 한다. 코드생성기를 만드는 도구의 입력인 기계묘사는 대체규칙(rewrite rule)들의 집합인 트리대체시스템(tree rewrite system)을 이용한다. 그리고 각 대체규칙에는 비용을 표시하는 수식이 있다.

수식을 대체규칙의 비용으로 이용하는 상향식 트리패턴 일치방법은 코드생성기의 가장 효과적인 기법이다. 상향식 트리패턴 일치방법은 다른 트리패턴 일치방법보다 더 빠르다[5, 10]. BURS(Bottom-Up Rewrite System)의 개발은 동적 프로그래밍을 코드생성기를 만드는 시간으로 옮기기 때문에 이 사실을 더욱 명백하게 하였다[9, 14]. 그러나 BURS는 패턴일치기를 생성하는데 많은 시간과 공간이 필요하고 트리대체시스템의 비용들은 상수라야 한다는 제약을 가지고 있다. BURS에 바탕을 둔 대표적인 코드생성기를 만드

† 정 회 원 : 전남대학교 정보통신학부 교수
논문접수 : 2004년 4월 15일, 심사완료 : 2005년 1월 24일

는 도구는 BURG이다[13]. 컴파일시간에 동적 프로그래밍을 수행하는 일치방법은 수식을 비용으로 허용한다. 수식을 비용으로 이용하면 처리할 수 있는 트리대체시스템들의 범위가 커지지만 컴파일 시간에 비효율을 유발할 수 있다. 이 방법은 기반을 둔 대표적인 코드생성기를 만드는 도구는 JBURG이다[16]. 본 논문에서는 수식을 비용으로 하는 상향식 트리패턴 일치 방법에 대해 연구를 수행하였다.

본 연구에서는 수식을 비용으로 이용하는 상향식 트리패턴 일치 방법에서 패턴일치기의 효율과 크기에 관해 다루었다. 패턴일치기의 효율을 향상시키기 위해 트리패턴들 사이의 관계를 분석하였다. 동적 프로그래밍을 이용하는 상향식 트리패턴 일치 방법은 중간표현 트리를 두 번 살펴본다. 첫 번째는 상향식으로 각 노드마다 상태를 계산한다. 상태는 주어진 트리패턴들로부터 계산된다. 두 번째는 트리를 하향식으로 살펴보며 최소비용커버를 찾아낸다. 그리면, 최소비용커버에서 목적코드가 생성되는 것이다. 기존의 패턴일치기는 상태를 계산하기 위해 노드에서 주어진 트리패턴들을 모두 사용하였다[6]. 그러나 트리패턴들 사이의 관계를 미리 알고 있으면 상태 계산을 위해 주어진 트리패턴들 중 일부만 이용해도 가능하다. 상태계산을 위해 사용될 트리패턴들을 찾기 위해 주어진 트리패턴들의 집합은 결정트리로 변환한다. 이 결정트리는 상태계산을 위해 필요한 트리패턴들 사이에 적용될 순서를 부여한 것이다.

2. 표기법과 기존연구

본 절에서는 트리, 패턴, 트리언어 그리고 트리대체시스템에 대한 표기법과 정의를 살펴보고, 여기서는 [9]에서 사용한 용어와 표기법을 주로 사용한다.

[정의 2.1] $\sum$는 터미널들의 집합으로서 중간표현을 나타낼 때 사용된다. 각 터미널은 양의 정수 값의 *arity*를 가진다. 이때 $\sum$를 기반으로 하는 트리언어는 다음과 같이 정의된다.

- *arity*가 0인 터미널 θ는 트리이다.
- $t_1, \ldots, t_n$가 트리이고 터미널 θ의 *arity*가 n이면 $\theta(t_1, \ldots, t_n)$는 트리이다.

트리의 부분트리는 위치라고 부르는 정수들의 나열에 의해 표기될 수 있다. 위치를 표기하는데는 "@"가 이용된다. 위치에 의한 부분트리 표기의 정의는 다음과 같다.

1. $\theta(t_1, \ldots, t_n)@_\varepsilon = \theta(t_1, \ldots, t_n)$.
2. $\theta(t_1, \ldots, t_n)@0 = \theta$, 이때 터미널 θ를 트리 $\theta(t_1, \ldots, t_n)$의 레벨이라 부른다.
3. $1 \le k \le n$이고 $@s$가 부분트리 t_k의 위치라면 $\theta(t_1, \ldots, t_n)@_{k \cdot s} = t_k@_s$이다.

[정의 2.2] 트리대체시스템은 G=(N, $\sum$, R, S)로 표기된다.

- N은 넌터미널들의 유한집합이다.
- $\sum$는 터미널들의 유한집합이다.
- R은 $\alpha \rightarrow A$형태인 대체규칙들의 유한집합이다. 여기서 α는 $\sum \cup N$을 기반으로 하는 트리로서 규칙의 입력패턴이라고 부르고 A는 넌터미널이다. 트리대체시스템의 각 규칙은 $c(\alpha \rightarrow A)$로 표기되는 비용을 위한 하나의 식을 가지고 있다. θ-patterns는 트리대체시스템에 나타난 입력패턴들 중에 레벨이 θ인 패턴들의 집합이다.

t를 $\sum \cup N$ 기반으로 하는 트리이고 $\alpha \rightarrow A$를 대체규칙이라 할 때 트리대체시스템에서 사용하는 여러 용어들을 설명한다. 부분트리 $t_{@p}$가 입력패턴인 α와 일치하면 α는 위치 p에서 트리 t와 일치한다고 말한다. 그리고 트리 t는 대체규칙 $\alpha \rightarrow A$에 의해서 새로운 트리 t'로 바뀐다. 이 바뀌는 과정을 $t \Rightarrow_{\alpha \rightarrow A} t'$로 표기되면 짧게는 $t \Rightarrow t'$라고 쓰인다. 트리대체시스템 G의 트리언어는 $L(G) = \{t | t \in T_\Sigma \text{ and } t \Rightarrow^* S\}$로 표기된다. 트리대체시스템에서 유도과정은 일반적인 스트링 대체시스템의 유도방향과 반대임을 주의해야 한다. 대체규칙들의 열을 대체열이라고 부른다.

$L(G)$의 하나의 트리 t에 대해서, $N \cup \sum$의 원소인 X에 대해 $t_{@p} \Rightarrow^* X$이면 대체열인 τ를 $t_{@p}$에서 X로서 커버들의 집합을 covers($t_{@p}$, X)에서 가장 싼 비용을 가진 커버들의 최소비용커버라 부르고 $LCV(t_{@p}, X)$로 표기한다. 여기서 사용되는 트리일치 알고리즘은 주어진 트리 t에 대해 $LCV(t, S)$를 발견하는 것이다.

$LCV(t, S)$를 계산하는 과정은 두 단계로 구성된다. 첫 단계에서는 중간표현트리를 상향식으로 방문하며 각 노드마다 상태를 계산한다. 상태는 (넌터미널, 규칙, 비용)으로 이루어진 아이템들의 집합이다. s가 $t_{@p}$의 루트노드에서 계산된 상태이고 $t_{@p}$의 레벨이 θ라면 다음과 같이 정의할 수 있다.

$s = \{(A, \text{rule}(t_{@p}, A), \text{cost}(t_{@p}, A)) | A \in N\}$, 여기서 $\text{rule}(t_{@p}, A)$는 트리 $t_{@p}$를 최소비용으로 넌터미널 A로 유도할 수 있는 대체열의 마지막 규칙이다. 그리고 $\text{cost}(t_{@p}, A)$는 최소비용 대체열의 비용이다. 그러므로 트리 $t_{@p}$를 넌터미널 A로 유도할 수 없으면 $\text{cost}(t_{@p}, A)$는 무한대(∞)를 값으로 가짐으로 유도할 수 없음을 나타낸다. $\text{cost}(t_{@p}, A)$는 주어진 트리패턴들을 (θ patterns) 모두 사용함으로 다음과 같이 계산할 수 있다 : $\text{cost}(t_{@p}, A) = \min\{\text{cost}(t_{@p}, \alpha) + c(\tau) | \forall \alpha \in \theta\text{-patterns}, \forall \tau : \alpha \Rightarrow_\tau A\}$. 두 번째 단계에서 패턴일치기는

하향식으로 중간표현트리를 살피면서 최소비용커버를 계산한다.

3. 상태를 효율적으로 계산하는 방법

기존 연구에서는 비용인 cost(t, A)를 θ-patterns의 모든 트리패턴들을 이용하여 계산하였다[6, 11, 12]. 그러나 트리패턴들 상의 관계들은 하나의 트리패턴을 이용한 결과는 다른 트리패턴들의 결과들을 결정할 수 있다는 사실을 말해준다. 이점이 연구의 동기이다.

[정의 3.1] α, β를 θ-patterns의 원소라고 하면, α와 β사이에 세 가지 관계가 다음과 같이 정의된다. T_Σ에 포함되는 모든 트리t에 대해 $t \Rightarrow^* \beta$일 때 $t \Rightarrow^* \alpha$이면 α≥β이다. 두 번째로 $t \Rightarrow^* \alpha$와 $t \Rightarrow^* \beta$를 만족하는 트리 t가 T_Σ에 존재하지 않으면 α ∥ β이다. 세 번째로 트리 t_1에 대해 $t_1 \Rightarrow^* \alpha$, $t_1 \not\Rightarrow^* \beta$이고, 트리 t_2에 대해 $t_2 \not\Rightarrow^* \alpha$, $t_2 \Rightarrow^* \beta$이고, 트리 t_3에 대해 $t_3 \not\Rightarrow^* \alpha$, $t_3 \Rightarrow^* \beta$를 만족하며 세 개의 트리 t_1, t_2, t_3이 T_Σ에 존재하는 α~β이다[5, 9].

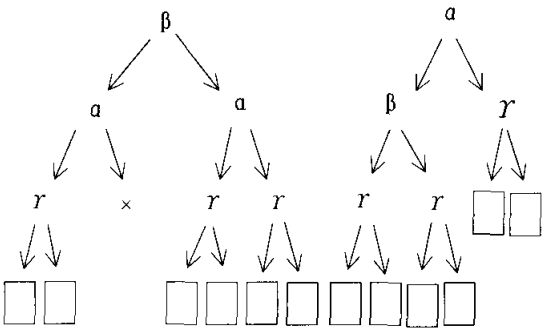
정의된 관계들을 이용하면 $cost(t, A)$는 효율적으로 계산이 가능하다. 계산은 다음과 같이 반복에 의해 결정된다.

(알고리즘 3.1) cost(t, A)를 **계산하는 알고리즘**

1. cost(t, A)=∞
2. set= θ-patterns
3. **for** α∈set **do**
4. Cost=cost(t, α)+min{c(τ)|α$\Rightarrow_\tau$ A}
5. **if** cost(t, α)=∞
6. **then** set=set-{ r | $r \in \theta$-patterns, α≥ r }
7. **else** set=set-{ r | $r \in \theta$-patterns, α ∥ r }∪{α}

집합 set은 for loop이 반복됨에 따라 공집합으로 가기 때문에 set에 더 이상 트리패턴이 남아 있지 않을 때 루프는 멈추게 된다. (알고리즘 3.1)의 3번째 줄에서 패턴 α의 선택은 루프의 반복횟수에 영향을 주고 있음을 주의해야 한다. 반복 횟수를 줄이기 위해 패턴 α를 어떻게 선택할 것인가에 대해 연구하였다.

본 연구의 접근 방식은 트리패턴들의 집합인 θ-patterns를 결정트리로 바꾸는 것이다. 결정트리의 각 노드는 θ-patterns에 속하는 한 트리패턴이다. 한 노드가 가지는 패턴이 α일 때 왼편가지는 cost(t, α)≠∞를 의미하고 오른편 가지는 cost(t, α)=∞)를 의미한다. (그림 1)은 θ-patterns의 보기와 가능한 결정트리들 중 두 개의 예를 보여주고 있다. 이 예에서 α가 β의 왼편 자노드일 때 cost(t, β)≠∞는 cost(t, α)≠∞를 포함하기 때문에 α의 오른편 자노드가 없음을 유의해야 한다.



(그림 1) θ-patterns와 결정트리의 예

결정트리를 평가하는데는 알고리즘의 루프가 끝나는 위치에 외부노드(external node)를 추가한다. (그림 1)에서는 사각형을 가지고 외부노드를 표시하고 있다. 결정트리의 외부경로길이(external path length)는 루트에서 각 외부노드로의 경로의 길이들을 합한 값이다. 이 값이 작은 결정트리가 알고리즘의 반복 횟수를 짧게 한다. 따라서 연구의 목표도 외부경로길이가 가장 작은 결정트리를 찾는 것이다. 하지만, 가장 작은 외부경로길이를 가지는 결정트리를 찾는 것은 NP-complete 문제이다.

[정리 3.1] 최소 외부경로길이 결정 트리를 찾는 것은 NP-complete 문제이다.

[증명] 만약 θ patterns(={α_1, α_2, …, α_l})가 같은 레벨을 가지는 패턴들의 집합이라면 외부노드의 개수는 2^l개를 넘지 않는다. 각 외부노드를 <v_1, v_2, …, v_l>라고 놓을 수 있다. 여기서 각 v_i는(for $1 \le i \le l$) 다음과 같이 정의된다.

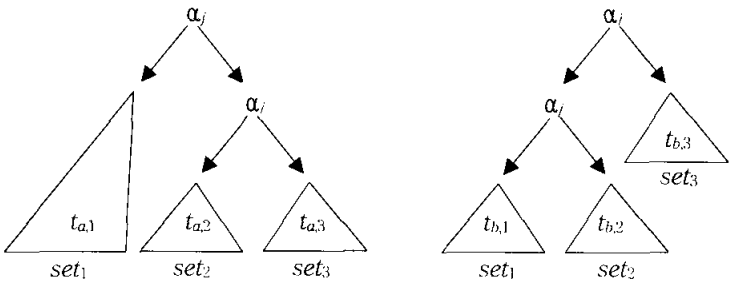
$$v_i = \begin{cases} \text{true} & \text{if } cost(t, \alpha_i) \neq \infty \\ \text{false} & \text{if } cost(t, \alpha_i) = \infty \end{cases}$$

만약 외부노드의 개수가 m이라면($m \le 2^l$) 최소 외부경로길이를 가지는 결정트리를 만드는 것은 m개의 레코드에 대한 최소접근시간을 가지는 trie를 만드는 문제와 동일하게 된다. 여기서 각 레코드는 이진 값을 가지는 l개의 필드로 이룬다. 이와 같은 trie를 결정하는 것은 NP-complete문제라고 이미 증명되었다[15].

[정리 3.2] 결정트리들이 θ-patterns에서 만들어지고 패턴 α, β가 θ-patterns의 원소라고 가정한다. α ≥ β이면 α가 β의 상위노드로 있는 결정트리가 β가 α의 상위노드로 있는 결정트리보다 외부경로길이가 작다.

[증명] θ-patterns(={α_1, α_2, …, α_l})가 같은 레벨을 가지

는 패턴들의 집합이고 $\alpha_i \geq \alpha_j$라고 가정하자. [정리 3.1]의 증명에서와 같이 외부노드들의 개수가 m이라고 하고 외부노드를 (v_1, v_2, …, v_l)라고 놓으면 외부노드들의 집합은 set_1, set_2, set_3으로 나눌 수 있다. set_1은 v_i =true and v_j =true인 외부노드들의 집합이고 set_2는 v_i =true and v_j =false인 노드들의 집합이고 set_3은 v_i =false and v_j =false인 노드들의 집합이다. 여기서 $\alpha_i \geq \alpha_j$이기 때문에 v_i =false and v_j =true인 외부노드들의 집합은 존재하지 않는다. 이와 같을 때 (그림 2)에서는 α_i, α_j 둘 사이의 테스트 순서에 따라 두 가지 가능한 결정트리를 보여주고 있다. (그림 2)에서 부분트리 $t_{a,2}$와 $t_{b,2}$는 같은 결정트리이다. 이것은 두 부분트리의 외부노드들의 집합이 set_2와 같기 때문이다. 마찬가지로 $t_{a,3}$과 $t_{b,3}$역시 같은 외부노드들의 집합 set_3 때문에 같다. 그러나 $t_{a,1}$과 $t_{b,1}$은 같은 외부노드들의 집합 set_1을 가지지만 서로 다르다. 만약 $cost(t, \alpha_j) \neq \infty$라고 한다면 $cost(t, \alpha_j)$와 $cost(t, \alpha_i)$는 둘 다 계산되어야 한다. (그림 2)의 (가)에서 $cost(t, \alpha_i) \neq \infty$이기 때문에 α_j에서 set_1의 외부노드로의 모든 경로들은 중간노드로 α_i를 포함한다. e_1, e_2, e_3을 $t_{a,1}$, $t_{a,2}$, $t_{a,3}$에 대한 외부경로길이라고 한다면 그림 2의 (가)의 외부경로길이는 $e_1+e_2+e_3$이고, (나)의 외부경로길이는 $e_1+e_2+e_3+|set_3|$이다.



(가) α_j를 α_i보다 먼저 테스트할 경우 (나) α_i를 α_j보다 먼저 테스트할 경우

(그림 2) 패턴에 대한 테스트 순서에 따른 결정 트리

(알고리즘 3.2)는 결정트리를 구하기 위해 고안한 알고리즘으로 [정리 3.2]를 이용하였다.

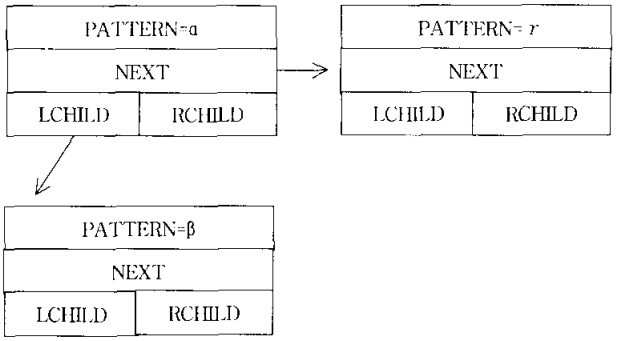
(알고리즘 3.2) 결정트리를 구하는 알고리즘 DT(set)

```
/* set은 트리패턴들의 집합을 위한 변수이다.
DT(set)은 set을 가지고 만들어진 결정트리이다. */

1. Anode=create_node()
   /* create_node()는 결정트리를 위한 새로운 노드를 만드는 함수
      이다. */
2. Anode.PATTERN=α
   where ∄β ∈ set such that β ≥ α
3. set_sub={β | α ≥ β, β ∈ set}
4. set_inc={β | α ∥ β, β ∈ set}
5. set_ind=set-set_sub-set_inc
6. Anode.LCHILD=DT(set_sub)
7. Anode.RCHILD=DT(set_inc)
8. Anode.NEXT=DT(set_ind)
9. return(Anode)
```

(알고리즘 3.2)에서 결정트리의 한 노드는 4개의 필드를 가진다; LCHILD, RCHILD, NEXT, PATTERN, LCHILD와 RCHILD는 왼편과 오른편 가지를 가진다. PATTERN은 노드를 표시하는 트리패턴이다. NEXT는 관계 ≥, ∥ 둘 다 만족하지 않는 두 패턴 사이에 연결을 위해 사용된다.

(그림 3)은 (그림 1)의 예를 가지고 (알고리즘 3.2)가 만들어낸 결정트리이다.



(그림 3) (알고리즘 3.2)의 결과로서 결정트리

(알고리즘 3.1)에서, 관계 ≥와 ∥의 계산은 많은 시간을 소모하는 작업이다. 이 문제는 코드생성기를 만드는 도구의 효율을 크게 떨어뜨릴 수 있다. 본 연구에서는 실제적인 기계묘사의 특징을 이용하여 관계 ≥와 ∥를 변형한 제한 조건을 제시한다. 이 제한 조건들은 실제 사용되는 기계묘사 상에서 관계 ≥와 ∥를 대부분 나타낼 수 있으며 쉽게 계산된다.

[정의 3.2] α와 β는 $T_{\Sigma,X}$의 원소인 트리패턴들이라고 가정

하고 다음 세가지 관계 $\geq_s$, $\|_s$, $\sim_s$를 정의한다.

- 아래의 조건들 중 하나를 만족하면 $\alpha \geq_s \beta$이다.
  - ($\alpha \in N$ or $\beta \in N$) and $\{ r \mid r \Rightarrow^* \beta,\ r$ 는 입력패턴$\} \subseteq \{ r \mid r \Rightarrow^* \alpha,\ r$ 는 입력패턴$\}$
  - $\alpha_{@0} = \beta_{@0} \in \sum$ and $\alpha_{@i} \geq_s \beta_{@i}$ for $1 \leq i \leq arity(\alpha_{@0})$
- 아래의 조건들 중 하나를 만족하면 $\alpha \|_s \beta$ 이다.

  ($\alpha \in N$ or $\beta \in N$) and $\{ r_{@0} \in \sum \mid r \Rightarrow^* \alpha,\ r$ 는 입력패턴$\} \cap \{ r_{@0} \in \sum \mid r \Rightarrow^* \beta,\ r$ 는 입력패턴$\} = \emptyset$,
  - $\alpha_{@0} = \beta_{@0} \in \sum$ and $\alpha_{@0} \neq \beta_{@0}$ or
  - $\alpha_{@0} = \beta_{@0} \in \sum$ 그리고 어떤 조건 i, j에 대해 $1 \leq i \leq arity(\alpha_{@0})$이고 $\alpha_{@i} \|_s \beta@i$
- 아래의 조건을 모두 만족하면 $\alpha \sim_s \beta$ 이다.
  1. $\alpha_{@0} = \beta_{@0} \in \sum$
  2. ($\alpha_{@i} \geq \beta_{@i}$ or $\beta_{@i} \geq \alpha@i$) for $1 \leq i \leq arity(\alpha_{@0})$
  3. 어떤 i, j에 대해 $1 \leq i, j \leq arity(\alpha_{@0})$이고 ($\alpha_{@i} \geq \beta_{@i}$ 그리고 $\beta_{@j} \geq \alpha_{@j}$)이다.

<표 1>은 관계 $\geq_s$, $\|_s$가 관계 $\geq$, $\|$ 을 실제적인 기계묘사에서 대부분 표현할 수 있음을 보여준다. 사용된 기계묘사들은 JBURG에서 사용된 것들이다.

〈표 1〉 기계묘사 상에서 $\geq_s$, $\|_s$, $\sim_s$ 와 $\geq$, $\|$, $\sim$

	x86	m68k	SPARC	MIPS
$>$, $\|$, $\sim$	289	323	49	32
$\geq_s$, $\|_s$, $\sim_s$	268	254	43	32

〈표 2〉 상태 생성에 사용되는 패턴 수

	x86		m68k		sparc	
	JBURG	본연구	JBURG	본연구	JBURG	본연구
8q.java	1170	795	847	579	500	416
array.java	1142	1065	1160	816	707	593
cf.java	552	453	536	422	358	310
cq.java	87427	79201	92861	78256	56289	49160
cvt.java	2655	1982	2862	1888	1907	1497
fields.java	1154	803	983	846	609	561
sort.java	1055	811	1134	761	604	484
struct.java	869	636	849	677	838	753
switc.java	2411	1743	2253	1732	1364	1169
front.java	163	121	208	169	79	68

<표 2>는 JBURG에서 만들어진 코드생성기와 본 연구에서 만들어진 코드생성기가 주어진 코드를 컴파일 할 때 패턴매칭을 몇 번씩 하는가에 관한 결과이다. <표 2>에서 보는바와 같이 약15%정도의 패턴을 사용하지 않아도 같은 코드 생성이 가능함을 알 수 있었다. 여기서 사용한 테스트 프로그램은 연구실 또는 인터넷에서 구할 수 있는 보통 프로그램이다.

결정트리를 이용하는 방법 이외에도 상태를 만드는데 사용되는 집합 θ-patterns에 원소가 하나만 존재할 때 적용될 수 있는 방법을 이용하면 더 성능향상을 얻을 수 있다. 상태의 정의로부터 각 넌터미널은 하나의 비용을 가지며 그 값은 절대값이 의미 있는 것이 아니고 다른 넌터미널의 비용에 대한 상대 값이 의미 있는 것이다. θ-patterns-$\{ r \}$ 일 때 $cost(t, A)$는 다음과 같이 계산할 수 있다. $cost(t, \beta) = \min\{c(\tau) \mid r \Rightarrow \tau\ \beta \}$. 이 방법은 속도뿐만 아니라 코드생성기 모듈을 작게 만들어 준다.

4. 결 론

본 연구에서는 코드생성시 동적 프로그래밍을 이용하는 상향식 트리패턴 일치에 시간 효율을 높이는 연구를 수행하였다. 제안된 방법은 기존의 방법보다 상태를 만드는데 있어 더 효율적이었다. 게다가 코드생성기를 더 작게 만들었다. 제안된 방법은 정형 적인 개념들을 가지고 정리하였다. 연구의 결과는 효율적인 JAVA 컴파일러를 만드는데 활용될 수 있기 때문에 JIT컴파일러 생성에 활용될 수 있다.

제안된 방법의 효과와 일반성의 정도는 기계묘사를 위한 트리대체시스템의 기술방법에 달려있다. 특히, 트리대체시스템이 같은 레벨을 가지는 입력패턴이 많을 때 그리고 패턴들간 관계가 제한 조건에 국한될 때 더 효율적인 코드생성을 할 수 있다. 이 밖에 요구되는 분석들은 코드생성기를 만드는 시간에 수행되며, 그것은 컴파일 시간에 효율을 좋게 한다는 점을 특징으로 한다.

참 고 문 헌

[1] A. Balachandran, D. M. Dhamdhere, S. Biswas, "Efficient retargetable code generation using bottom-up tree patterns matching", Computer Languages Vol.15, No.3, pp.127-140, 1990.

[2] A. V. Aho, M. Ganapathi, S. W. K. Tjiang, "Code Generatin Using Tree Matching and Dynamic Programming", ACM TOPLAS Vol.11, No.4, pp.159-175, 1989.

[3] A. V. Aho, R. Sethi, J. K. Ullman, "Compilers -Principles, Techniques, and Tools", Addison Wesley, 1986.

[4] C. W. Fraser, David R. Hanson, "A Retargetable C Compiler : Design and Implementation", The Ben-jamin/Cummings, 1995.
[5] C. M. Hoffmann, M. j. O'Donnell, "Pattern Matching in Trees", ACM Journal, Vol.29, No.1, pp.68-95, 1982.
[6] C. W. Fraser, D. R. Hanson, T. A. Proebsting, "Engineering a simple, Efficient Code Generator Generator", ACM LOPLAS, Vol.1, No.3, pp.331-340, 1992.
[7] C. W. Fraser, R. R. Henry, T. A. Proebsting, "BURG-Fast Optimal Instruction Selectin and Tree Parsing", ACM SIGPLAN Nocices, Vol.27, No.4, pp.68-76, 1991.
[8] D. R. Chase, "An improvement to bottom-up tree pattern matching", 14th Annual symposium on POPL, pp.168-177, 1987.
[9] E. Pelegri-Liopart, "Rewrite Systems, Parrern Matching, and Code Generation", PhD Dissertation, Report No. UCB/CSD 84/184, CSD, EECS, UCB, CA, May 1988.
[10] E. Pelegri-Liopart, S. L. Graham, "Optimal Code Generation for Expression Trees : An Application of BURS Theory", 15th Annual ACM SIGACT-SIGPLAN symposium on POPL, San Diego, California, pp.294-307, January, 1988
[11] F. Guillaume, L. George, "MLBurg-Documentation", 1993.
[12] K. John Gough, "Bottom-Up Tree Rewriting Tool MBURG", ACM SIGPLAN Notices, Vol.31, No.1, pp.28 31, 1996.
[13] T. A. Preobsting, "Simple and efficient BURS table generation", SIGPLAN Notices, Vol.27, No.7, pp.331-340, 1992.
[14] T. A. Proebsting, C. Fischer, "Code Generation Techniques", PhD Dissertation, Department of Computer Science, University of Wisconsin-Madison, 1992.
[15] D. Comer, R. Sethi, "The Complexity of Trie Index Construction", ACM Journal, Vol.24, No.3, pp.428-440, 1977.
[16] T. Harword, "Announce : JBURG, a Java-based BURG", http://compilers.iecc.com, July, 2002.
[17] Soo-Mook Moon외 9인, "LaTTe : a Java VM just in-time compiler with fast and efficient register allocation", International Conference on Parallel Architectures and Compilation Techniques, pp.128-138, March, 2000.

강 경 우

email : kwkang@cheonan.ac.kr
1990년 경성대학교 전산통계학과(이학사)
1992년 한국과학기술원 전산학과(공학석사)
1998년 한국과학기술원 전산학과(공학박사)
1998년~1999년 한국과학기술정보연구원 선임연구원
2000년~현재 천안대학교 정보통신학부 조교수
관심분야 : 컴파일러, 그리드 컴퓨팅, 유비쿼터스 컴퓨팅 등