# 흐름 그래프 형태를 이용한 함수형 프로그램 유사성 비교

# (A Program Similarity Check by Flow Graphs of Functional Programs)

서 선 애 † 한 태 숙 ††
(Sunae Seo) (Taisook Han)

**요 약** 컴퓨터와 소프트웨어의 사용이 증가하면서, 프로그램 소스의 도용(표절)이 사회적인 문제로 부각되고 있다. 이런 문제를 해결하고자 프로그램의 문법 구조를 비교하여 표절을 찾아내는 방법론이 제안되었지만, 간단한 프로그램 수정에도 표절을 찾아내지 못하는 한계를 가지고 있다. 이 연구에서는, 문법 구조적인 정보 뿐 아니라, 프로그램식 간의 수행시 의존 관계를 드러내는 그래프를 이용한 프로그램 표절 감지 시스템을 제안한다. 이 방법론은 문법 정보 뿐 아니라, 수행시 의존 관계까지 비교 대상에 올림으로써, 수행시 의존 관계를 변화시키지 못하는 프로그램 수정에 대해서도 프로그램 표절을 판별할 수 있다. 또한, 이 연구에서는 표절 프로그램이란 무엇인가를 엄밀하게 정의하고 이 표절 프로그램의 정의와 연구에서 제안된 표절 감별 그래프와의 관계를 보였다. 즉, 두 프로그램이 표절이라는 것은 표절 감별 그래프가 일치한다는 것과 필요 충분 관계가 있음을 증명하였다. 또한 제안된 표절 감별 방법론을 실제적인 프로그래밍 언어인 nML 에 대해서 구현하였다. 구현된 도구를 통해서 실제 표절된 프로그램들을 감별한 결과, 기존의 방법에서 찾기 어려운 프로그램 표절을 제안된 방법론이 다룰 수 있음을 확인하였다.

**키워드** 프로그램 유사성, 표절 프로그램 검사, 흐름 분석 그래프, 프로그램 분석, 함수형 언어

***Abstract*** Stealing the source code of a program is a serious problem not only in a moral sense but also in a legal sense. However, it is not clear whether the code of a program is copied from another or not. There was a program similarity checker detecting code-copy by comparing the abstract syntax trees of programs. However this method has a limitation that it cannot detect the code-copy attacks when the attacker modifies the syntax of the program on purpose. We propose a program similarity check by program control graph, which reveals not only syntax information but also control dependancy. Our method can detect the code-copy attacks that do not change control dependancy. Moreover, we define what code-copy means and establish the connection between code-copy and similarity of program control graph: we prove that two programs are related by copy congruence if and only if the program control graphs of these programs are equivalent. We implemented our method on a functional programming language, nML. The experimental results show us that the suggested method can detect code similarity that is not detected by the existing method.

**Key words** : program similarity, code-copy detection, control flow graph, program analysis, functional programming language

## 1. 서 론

프로그램 도용(표절)은 도덕적인 이유뿐만 아니라 법적인 이유로도 심각한 문제이다. 그러나 프로그램의 표절을 명확하게 정의할 수 없기 때문에 프로그램 도용을 찾아내기에는 어려움이 많다. 프로그램 표절을 프로그램의 의미론(semantics)적으로 해석하면 같은 일을 하는 모든 프로그램이 표절이라고 판정될 수 있다. 예를 들어, 서로 다른 방법으로 정수를 정렬하는 두 정렬 함수를 표절되었다고 말할 수는 없지만 의미론적으로 해석하면 같은 프로그램이다. 반대로 문법(syntax)적으로 동

· 본 연구는 첨단정보기술 연구센터를 통하여 과학재단의 지원을 받았음
† 학생회원 : 한국과학기술원 전산학과 박사과정
saseo@ropas.kaist.ac.kr
†† 종신회원 : 한국과학기술원 전산학과 교수
han@pllab.kaist.ac.kr
논문접수 : 2004년 10월 13일
심사완료 : 2005년 2월 22일

일한 프로그램을 표절이라고 정의하면 간단한 수정과정을 거치기만 하면 표절된 프로그램도 표절되지 않았다고 할 수 있다.

문법만을 기준으로 표절을 찾아내는 방법론은 자동적으로 검사할 수 있지만, 간단한 프로그램 수정에는 표절을 판별할 수 없는 한계가 있다. 문법 구조를 바탕으로 프로그램 표절을 찾아내는 검증기인 CloneChecker[1]는 실제로 교육현장에서 실험해 본 결과 프로그램 표절 감지에 좋은 효과를 발휘하였으나, 몇 가지 한계를 노출하였다: (1) "프로그램 표절"의 의미가 문법적인 형태에 한정되어 있어서, 단순한 프로그램 변형에는 효과적이지 못하였으며, (2) 개발된 도구가 감지할 수 있는 "표절 프로그램"의 한계에 대해서 명확한 제시를 하지 못하였다.

**예제 1.** 다음 두 프로그램은 문법을 기준으로 표절을 판단할 수 없는 것들이다.

```
let tc =  λx.  let m = λy. e1
               in
                m  x
               end
in
   tc
end
```

과

```
let m = λy. e1 in
let tc = λx. m x in
 tc
end
```

왼쪽 프로그램에서 $m$의 정의를 $tc$ 정의 밖으로 꺼내는 단순한 변형을 가한 것이 오른쪽 프로그램이다. 의미적으로 위의 두 프로그램은 같은 것이지만, 문법적인 방법으로는 표절을 감지할 수 없다. 왜냐하면, 위의 두 프로그램은 서로 다른 문법 구조를 가지고 있기 때문이다. 프로그램 표절을 단순히 문법 구조에 대해서만 정의한다면, 예제에서 보이는 것과 같이 간단하고 의심이 될 만한 프로그램 변형에 대해서 문법적인 방법론은 표절 감지를 할 수 없다. □

위와 같은 예제뿐만 아니라, 다음과 같은 공격에 대해 문법만을 기준으로 표절을 감별하기 어렵다:

(1) 유효범위(scope)를 변화시키는 **let**이나 **fix** 등을 넣거나 빼으로서 변형한 프로그램 표절(예제 1 의 경우),
(2) 변수의 이름 바꾸기,
(3) 연산의 교환 법칙 등을 응용한 프로그램 코드의 이동,
(4) 컴파일 시에 계산 가능한 부분의 부분적인 계산, 그리고
(5) 프로그램 수행이 닿지 않는 곳에 임의 코드 삽입.

이와 같은 경우들 중에 (1)의 경우는 변수와 값 사이의 바인딩(binding) 정보가 추가된다면 쉽게 찾아낼 수 있다. 이런 바인딩 정보는 프로그램식 간의 수행시의 의존 관계(control dependancy)와 관련이 깊다. 또한 이런 수행시의 의존 관계는 (5)의 경우 중의 일부에 해당하는 프로그램 표절을 찾아낼 수 있게 도움을 준다. 결국, 프로그램식 간의 수행시의 의존 관계를 살펴보는 것은 더 많은 프로그램 표절을 찾아내는 데에 큰 도움을 준다.

이 연구에서는, 문법의 구조적인 정보 뿐 아니라, 프로그램식 간의 수행시 의존 관계도 드러내는 그래프를 이용한 프로그램 표절 감지 시스템을 제안한다. 연구에서 제안하는 표절 감별 그래프를 이용하면 기존의 연구에서 감별하지 못하는 프로그램 표절을 감별해 낼 수 있다. 이 연구에서는 "프로그램 표절"를 명확히 정의하고, "프로그램 표절"과 "표절 감별 그래프의 일치"에 대해서 이론적인 맥을 세우고자 한다.

논문의 나머지 부분은 다음과 같이 구성된다: 2장에서는 기존의 표절 검사 방법론에 대해서 간략히 소개한다. 3장에서는 간략화 된 프로그램의 문법 구조를 소개한다. 프로그램 표절의 정의는 "표절 관계"라는 것을 통해 4장에서 소개된다. 프로그램의 표절 감별 그래프에 관한 정의와 프로그램에서 그래프로 변환하는 과정은 5장에서 설명된다. 또, 5장에서는 표절 감별 그래프에서 "일치"라는 것이 무엇인지를 명확히 정의한다. 6장에서는 프로그램 표절과 표절 감별 그래프의 일치 간에 필요 충분 관계가 있음을 보인다. 7장에서는 제안된 방법론의 구현과 실험에 대해서 소개하고, 8장에서 결론을 맺는다.

논문에서 정의한 정리와 보조정리들의 증명은 논문의 영어본[2]에서 찾을 수 있다.

## 2. 기존의 표절 검사 방법론

프로그램이나 문서의 표절이 여러 가지 문제점을 일으키고 있는 만큼 표절 방법이나 표절을 검사하는 방법론에 관한 연구가 여러 방면에서 진행되고 있다[1,3-7].

문서의 표절 검사 방법으로 가장 널리 사용되는 방법은 많이 사용되는 유사한 단어들이나 키워드들의 사용 횟수를 바탕으로 문서의 표절 여부를 판단하는 지문법(finger print)이다. 일반적인 문서의 비교 뿐 아니라, 프로그램 코드에 대해서도 지문법을 사용할 수 있는데, SIM(Software Similarity Tester)[3]가 그 예이다.

프로그램 소스 코드는 한정된 문법 구조를 가지기 때문에 지문법을 좀 더 확장한 표절 검사를 사용할 수 있다. 한정된 문법 구조로 인해서, 프로그램 소스 코드의 토큰들은 구조적인 형태를 가지게 되고, 그 형태를 여러 가지 비교 방법으로 비교하여 유사성을 계산할 수 있다.

YAP[4], MOSS[5], JPlag[6]등과 같은 시스템들은 프로그램 소스를 적절한 토큰의 리스트로 바꾸고, 그 리스트 사이의 유사성을 여러 가지 문자열 매칭 방법으로 찾아낸다. CloneChecker[1]는 두 프로그램의 문법 트리들의 자식 트리들을 모두 비교하여 같은 자식 트리의 개수를 계산함으로써, 문자열 매칭과는 다른 표절 검사 방법을 사용한다.

최근에 발표된 유전체 서열의 정렬 기법을 이용한 표절 검사[7,8]는 요즘 널리 연구되고 있는 유전자 염기 서열 비교 방법론을 표절 검사에 응용하였다. 이 방법은 프로그램의 키워드들을 아미노산에 매핑하여 염기 서열들을 검사하는 방법으로 표절 검사를 수행한다. 또한, 프로그램 흐름을 예측해서 흐름 순서대로 키워드들을 배치하여서, 프로그램 순서를 바꾸거나 안 쓰이는 코드를 삽입한 표절을 잘 검사할 수 있었다. [8]에서는 서열의 유사도를 가장 유사한 한 부분에 대해서만 계산하는 것이 아니라, 서열의 여러 부분에서도 재귀적으로 같은 방법을 적용시킴으로 기존의 방법을 개선하였다.

본 연구는 기존의 연구들에서 벗어나지 못한 문법 구조 기반의 비교를 벗어나서, 실제 프로그램 흐름 정보를 드러내는 새로운 구조를 제안한다. 연구에서 사용되는 흐름 정보는 프로그램 분석을 이용한 정확한 흐름 분석이므로 [7]에서 사용한 프로그램 흐름 예측과 매우 다르다. 또 제안된 방법에서 표절로 드러난 프로그램들은 특별한 관계에 있음을 보임으로 제안된 방법이 검사 가능한 표절의 한계를 분명히 하였다는 데에 의미가 있다. 프로그램 흐름 정보를 바꾸지 못하는 표절의 특성을 이용한 이 새로운 구조는 기존의 구조 비교 방법론들과 함께 사용될 수 있으므로, 기존의 표절 검사의 변별력을 한층 높일 수 있다. 논문의 7장에서 CloneChecker의 방법과 그 방법론을 우리의 방법과 함께 사용한 실험 결과를 보인다. 이 실험 결과는 우리의 방법론을 기존의 방법과 함께 사용하였을 때, 서로 상호 보완적인 효과를 가짐을 보이고 있다.

## 3. 간략화된 프로그램 문법 구조

우리가 대상으로 한 프로그래밍 언어는 모든 상위 레벨 언어의 핵심 언어로서 그림 1과 같은 문법 구조를 가진다. 그림 1에서 $v$는 정수나 실수 같은 상수를 말하고, $x$는 프로그램 변수이다. $\lambda x.e$는 $x$를 인자(parameter)로, $e$를 몸체(body)로 하는 함수이다. **fix** $f\ \lambda x.e$는 재귀 함수를 말하는데, $f$는 함수 몸체 $e$에서 자기를 부를 때 사용되는 이름이고 $x$는 인자이다. **let** $x = e_1$ **in** $e_2$ **end**는 프로그램식 $e_2$의 내부에서 프로그램식 $e_1$을 $x$라는 이름으로 사용한다는 의미를 가진다. 마지막으로, $e_1 + e_2$는 덧셈을 말한다.

```
e  ::=  v | x
      | λx.e | fix f λx.e
      | let x = e in e end | e e
      | e + e
p  ::=  e
```

그림 1 프로그램의 문법 구조

간략화 된 이 문법은 표절을 시도하기에 편리한 점을 충분히 갖추고 있다. 변수는 변수 이름 바꾸기 표절 시도에 이용될 수 있고, **let**이나 **fix** 프로그램식은 유효 범위(scope)를 변형하여 표절을 시도하는 공격에 이용되기 쉽다. 또한 더하기 연산자 +는 교환 법칙이 가능한 연산자이므로 교환 법칙을 이용한 표절이 가능하다.

이 연구에서는 위의 프로그램 문법을 De Bruijn 표기법[9]을 포함하는 문법 구조로 변형하여 사용한다. 변형된 문법은 함수의 인자 대신 De Bruijn 표기법을 사용한다. 변형된 문법 구조를 사용하는 데에는 두 가지 이유가 있다. 첫째로, De Bruijn 표기법은 함수의 인자를 변수로 사용하지 않고 숫자를 쓰므로, 변수 이름 바꾸기 표절을 막을 수 있다. 두 번째로, 함수 인자를 De Bruijn 표기법으로 바꾸지 않으면 후에 표절 프로그램과 표절 감별 그래프 간의 안전성을 만족시키지 못하는 경우가 생기기 때문이다.

De Bruijn 표기법을 포함하는 변형된 프로그램 문법은 그림 2와 같다. 이 문법 구조는 두 가지 가정을 한다: (1) 모든 프로그램식은 유일한 레이블을 가진다, (2) 모든 프로그램 변수는 서로 다르다.

```
e  ::=  v | x | d
      | λe^l | fix f λe^l
      | let x = e^l in e^l end | e^l e^l
      | e^l + e^l
p  ::=  e^l
```

그림 2 변환된 프로그램의 문법 구조

예제 1의 프로그램들을 레이블과 De Bruijn 표기법을 사용하는 프로그램으로 바꾸면 예제 2와 같다.

**예제 2.** 예제 1에서 레이블과 De Bruijn 표기법을 사용해서 바꾼 프로그램이다.

```
let tc = λ^l3. let m = λ^l9. e_1^l10
               in
                 (m^l7 0_d^l8)^l5,l6
               end^l4
in
  tc^l2                                  과
end^l1
```

$$\begin{array}{l}\texttt{let } m = \lambda^{l_9}.\ e_1^{l_{10}} \texttt{ in}\\ \texttt{let } tc = \lambda^{l_4}.\ (m^{l_7}\ 0_d^{l_8})^{l_6,l_5} \texttt{ in}\\ \quad tc^{l_3}\\ \texttt{end}^{l_1,l_2}\end{array}$$

## 4. 프로그램 표절

표절 감별 알고리즘을 정의하기 전에 프로그램 표절에 관해 직관적으로 정의한다. 우리가 표절된 두 프로그램이라고 말할 때, 그것은 두 프로그램의 문법 구조가 같은 것을 의미하지는 않는다. 이 연구에서 두 프로그램이 표절되었다는 것은 두 프로그램이 표절 관계에 있다는 것과 같다. 표절 관계는 간단하고 직관적인 프로그램 변경을 포함하고 있다. 그림 3에서는 프로그램의 문맥을 정의하고 이어서 표절 관계를 정의한다.

문맥은 프로그램의 문법 구조에 따라 정의가 되며, 표절 관계는 프로그램 표절 시도를 위한 프로그램 변형과 연관이 있다. 만일 두 프로그램이 완전히 같다면 그들은 표절 관계에 있다(taut). 교환 법칙이 성립하는 연산자의 두 피연산자를 바꾸는 것은 프로그램 표절이라고 말할 수 있다(com). **fix** 함수의 이름을 바꾸는 것이나(fix), 재귀적이지 않은 함수를 재귀적인 함수로 바꾸는 것(fixrm)은 프로그램 표절로 볼 수 있다. **let** 프로그램식에서 변수를 사용하지 않고 식을 직접 끼워 넣는 것도 프로그램 표절로 볼 수 있다(beta). 프로그램의 일부가 표절되었고 나머지가 같다면 표절한 것으로 볼 수 있고(context), 마지막으로, 표절한 것을 표절하면 또한 표절로 볼 수 있다(tr).

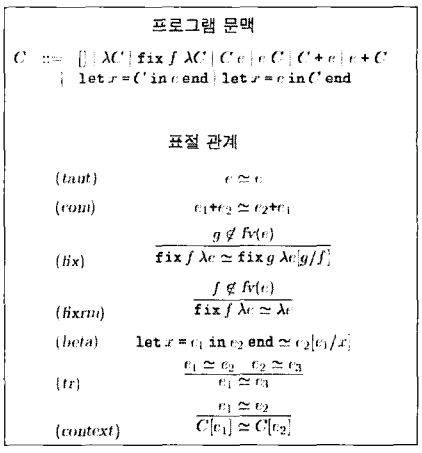


그림 3 프로그램 문맥과 표절 관계

1장에서 프로그램의 문법을 정의할 때, 모든 변수들과 레이블은 다르다고 정의를 하였으므로, 위의 표절 관계에서 치환( $[e_1/x]$)은 조심스럽게 정의되어야 한다. 변수 $x$가 나타나는 모든 곳에 $e_1$을 대신 삽입할 때, 매 경우마다 $e_1$내에 존재하는 변수들과 레이블을 모두 새로운 것으로 바꾸어 주어야 하기 때문이다. 치환의 엄밀한 정의는 아래와 같다.

**정의** 1. (치환)

$$e[e_0/x] = \forall x^{l_i}.\ e'[(e_0^{l_i})/x^{l_i}].$$

단, $x^{l_i}$는 프로그램식 $e^l$에 나타나는 변수들이고, $e_0$, 로 치환할 때 $e_0$, 속의 묶인(bound) 변수들과 레이블들은 새로운 변수들과 새로운 레이블로 바뀌어 있다고 가정한다. □

예를 들어, 프로그램식 $(x^{l_1}+x^{l_2})[(\lambda y.y^{l_3})/x]$은 치환을 하고 나면, $(\lambda z.z^{l_3})^{l_1}+(\lambda w.w^{l_3})^{l_2}$으로 바뀐다.

## 5. 표절 감별 그래프

이 장에서는 표절 감별 그래프를 정의한다. 우선, 프로그램식 간의 수행시 의존 구조를 드러내는 그래프와 프로그램에서 그래프로 변환을 해 주는 변환 함수를 정의한다. 다음으로, 그래프 줄이기 함수를 정의하고, 표절 감별 그래프를 정의한다. 마지막으로, 두 표절 감별 그래프가 같다는 것을 정의한다.

### 5.1 그래프

표절 감별을 위해 사용되는 그래프는 프로그램의 문법 구조 뿐 아니라, 바인딩 정보까지도 포함한다. 노드(node)는 문법 구조를 나타내는 구성자로 되어 있고, 연결선(arc)은 각 구성자들 간의 수행시 의존 관계를 나타낸다.

| | | | |
|---|---|---|---|
| *Graph* | $\mathcal{G}$ | ::= | $\mathcal{P}(Arc)$ |
| *Label* | $l$ | | |
| *Variable* | $x$ | | |
| *DBIndex* | $d$ | | |
| *LabVar* | $s$ | ::= | $x \mid l$ |
| *Node* | $n$ | ::= | $s \mid d \mid \lambda(l) \mid \mathbb{C}(v) \mid +(l,l) \mid @(l,l)$ |
| *Arc* | $a$ | ::= | $s \rightarrow n$ |

그림 4 그래프의 문법 구조

그림 4에 보이는 것과 같이, 문법 구조적으로 그래프는 연결선들의 집합이다. 각 연결선의 시작 노드는 레이블 노드 또는 변수 노드이고, 끝 노드는 모든 종류의 노드들이 올 수 있다. 연결선 $s_1 \rightarrow n_1$는 $s_1$이 $n_1$에 의존한다는 것을 의미한다.

### 5.1 그래프로의 변환

프로그램을 그래프로 변환하는 과정은 함수 T를 통해서 이루어진다. 그림 5에 정의된 함수 T는 프로그램식을 받아서 그래프를 준다. 프로그램식의 문법 구조에 따라 그래프의 노드가 정의되고, 프로그램식들 간의 수행 의존 관계에 따라 연결선이 생성된다. 특히, **fix**의 경

우 재귀 함수 이름 $f$는 자기 자신을 의미하므로, $f \rightarrow l$의 연결선이 필요하고, **let**의 경우 변수 $x$는 프로그램식 $e_1^{l_1}$의 값에 의존하므로 $x \rightarrow l_1$의 연결선이 필요하다. 예제 2를 변환한 그래프는 그림 6과 같다. 오른편 아래에 d라는 기호를 가진 숫자 0은 DeBruijn 색인을 의미한다.

그림 6을 보면, 두 프로그램으로부터 생성된 그래프의 모양이 레이블의 순서 관계를 제외하고 거의 유사함을 알 수 있다. 그림 6의 왼쪽 그래프를 살펴보면, 제일 첫 노드인 $l_1$에서 의미 있는 문법 구성자인 $\lambda l_4$까지 도달하려면 네 개의 노드를 지나야 한다:

$$\{l_1 \rightarrow l_2,\ l_2 \rightarrow tc,\ tc \rightarrow l_3,\ l_3 \rightarrow \lambda l_4\}.$$

오른쪽 그래프의 경우에는 $l_1$에서 다섯 개의 노드를 지나면, $\lambda l_5$를 만난다:

$$\{l_1 \rightarrow l_2,\ l_2 \rightarrow l_3,\ l_3 \rightarrow tc,\ tc \rightarrow l_4,\ l_4 \rightarrow \lambda l_5\}.$$

명료하게 양쪽의 레이블 $l_1$은 같은 문법 구성자를 지닌 노드에 의존한다. 이런 의존 관계를 극대화 시켜서 표절 감별을 하기 위해서는 이 그래프들을 다시 변형할 필요가 있다.

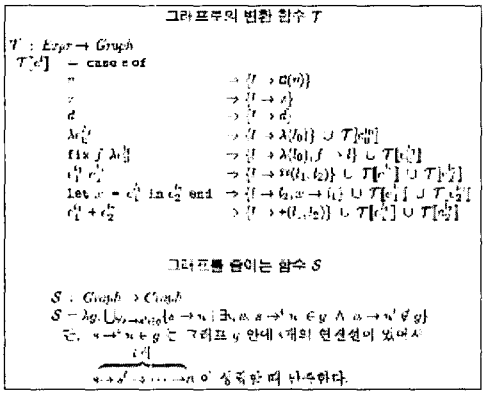


그림 5 그래프를 위한 함수들

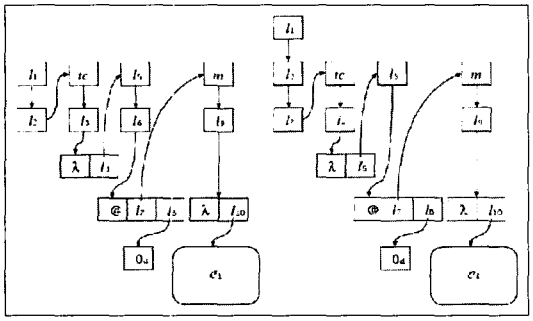

그림 6 변환 함수 T를 이용해서 그래프로 변환시킨 예제 1의 프로그램들

### 5.3 그래프 줄이기

변환 함수로부터 생성된 그래프에서 수행 흐름 정보를 살리면서 프로그램의 변형에 강한 그래프를 만들기 위해서는 한 줄의 연결선으로 의존 관계가 나타나는 노드들을 없애주면 된다. 이런 노드들을 없애는 것은 수행 시 의존 관계가 같으면서 서로 다르게 변형된 프로그램들을 한 가지 형태로 바꿈으로 표절 감별을 극대화 시킨다. 즉, 연결선에 의한 의존 구조를 단순화함으로써, 오히려 의존 구조를 명확히 한다.

예를 들어, 그림 6의 왼쪽 그래프에서 한 줄의 의존 관계가 있는 연결선들은 다음과 같다:

$$\{l_1 \rightarrow l_2, l_2 \rightarrow tc, tc \rightarrow l_3, l_3 \rightarrow \lambda l_4\},$$
$$\{l_4 \rightarrow l_5, l_5 \rightarrow l_6, l_6 \rightarrow @(l_7, l_8)\},$$
$$\{l_7 \rightarrow m, m \rightarrow l_9, l_9 \rightarrow \lambda l_{10}\},\ \{l_8 \rightarrow 0_d\}.$$

그래프 줄이기 과정을 통해서 이런 연결선들을 간략화시킬 수 있다. 즉, 위의 연결선 집합들은 다음과 같이 간략화된다:

$$\{l_1 \rightarrow \lambda l_4\},\ \{l_4 \rightarrow @(l_7, l_8)\},\ \{l_7 \rightarrow \lambda l_{10}\},\ \{l_8 \rightarrow 0_d\}.$$

이와 같은 그래프 줄이기 함수 S를 그림 5와 같이 정의할 수 있다.

이 그래프 줄이기 과정을 거치고 나면 모든 시작 노드들은 나가는 연결선이 없는 끝 노드로 직접 연결된다. 즉, 각 노드들은 프로그램의 수행 시 의존 관계가 존재하는 다음 프로그램 구성자 노드들로 직접 연결이 된다. 예제 1의 그래프로부터 그래프 줄이기 과정을 마친 그래프는 그림 7과 같다. 두 프로그램으로부터 만들어진 그래프는 그래프 줄이기를 거치면 같은 모양이 됨을 알 수 있다.

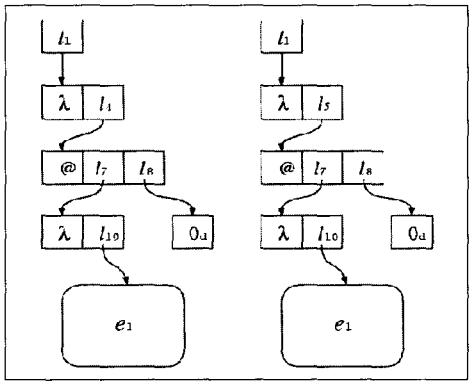

그림 7 줄이기 함수 S를 이용해 그림 6에서 줄인 그래프

### 5.4 표절 감별 그래프의 일치

이 절에서는 "표절 감별 그래프"라고 이름 붙인 최종적인 프로그램 그래프를 정의하고, 두 그래프가 같다는

것을 이론적으로 정의한다.

주어진 프로그램 $P$의 표절 감별 그래프 $R(P)$는 다음과 같이 정의된다.

**정의 2.** $R(P) \triangleq |\mathcal{S} \circ \mathcal{T}[\![e^l]\!]|_l$.

여기서, $|\ |_l$은 $l$에서부터 연결선을 통해서 닿을 수 있는 부분 그래프를 말한다.

**정의 3.** $|G|_s \triangleq (lfp\ F_G)\ (s)$,

단,

$$F_G = \lambda f.\lambda s.\ for\ s \to s' \in G.\ \{s \to s'\} \cup \left( \begin{array}{lcl} case\ s'\ of & & \\ \lambda(l) & \Rightarrow & f(l) \\ @(l_1, l_2) & \Rightarrow & f(l_1) \cup f(l_2) \\ +(l_1, l_2) & \Rightarrow & f(l_1) \cup f(l_2) \end{array} \right).$$

$|\ |_l$ 로 정의되는 닿을 수 있는 부분 그래프는 재귀적으로 정의된다. 함수에서 lfp는 최소고정점(least fixed point)을 의미하는 수학적 용어이다. $|\ |_l$을 이용해서 닿을 수 있는 부분 그래프만을 표절 감별 그래프로 정의하는 것은 몇 가지 이점이 있다: (1) 부분 그래프는 원래의 그래프보다 크기가 작으므로 그래프 비교 비용을 줄일 수 있고, (2) 부분 그래프에 포함되지 않은 그래프의 다른 부분은 실제 프로그램의 수행 시 수행할 수 없는 죽은 코드(dead code)를 의미하므로, 죽은 코드를 삽입하는 프로그램 표절 시도를 막을 수 있다.[1)]

정의 2에서 정의되는 표절 감별 그래프는 여러 과정을 통해서 생성되므로, 그 과정들에 관계된 특별한 성질을 가지게 된다.

**사실 1** 모든 프로그램 P 에 대하여 R(P) 는 다음의 모양을 갖는다:

$$\{l \to n \mid n \in \{x, d,\ \mathtt{C}(v),\ @(l, l), \lambda l,\ +(l, l)\}\}$$

그래프 줄이기 이후에 그래프의 모든 시작 노드들은 나가는 연결선이 없는 끝 노드로 직접 연결된다는 성질은 앞에서도 언급한 바가 있다. 더불어, 부분 그래프로 축소된 표절 감별 그래프는 전역 변수를 제외한 어떠한 변수도 가지고 있지 않게 된다.

**사실 1의 증명.** 다음의 두 가지에서 사실 1은 증명이 된다.

첫 번째는 그래프 줄이기 과정 이후에 들어오는 연결선을 가진 레이블 노드는 전혀 없다. 변환 함수를 살펴보면, 들어오는 연결선을 가진 레이블은 세 가지(**let**에서 두 개, **fix**에서 한 개)라는 것을 알 수 있다. 모든 레이블은 프로그램식에 매달려 있으므로, 레이블에서 프로그램식을 나타내는 노드 구성자로 가는 연결선이 반드시 존재한다. 즉, 들어오는 연결선이 있는 레이블들은 나가는 연결선도 존재하므로, 그래프 줄이기 과정에서 사라진다. 따라서, 그래프 줄이기 이후에 들어오는 연결선이 있는 레이블은 없다.

두 번째는 그래프 줄이기 이후에 나가는 연결선을 가진 변수 노드는 존재하지 않는다. 나가는 연결선을 가지는 변수는 두 종류가 있음을 변환 함수 정의에서 알 수 있다. 하나는 **let** 프로그램식의 변수이고, 다른 하나는 **fix** 함수의 이름이다. 이 변수들은 **let** 프로그램식이나 **fix** 프로그램식의 의미에 따라, 다음에 오는 프로그램식 내에서 사용이 되거나 혹은 사용이 되지 않는 두 경우만을 가지게 된다. 만일 사용이 된다면, 각 변수 노드들은 들어오는 연결선을 가지게 되고, 이미 존재하는 나가는 연결선과 함께 그래프 줄이기에서 사라진다. 만일 사용이 되지 않는다면, 프로그램 수행이 닿지 않는 죽은 코드가 되어서, $|\ |_l$ 의 정의에 따라 표절 감별 그래프의 l 에서 닿을 수 있는 부분 그래프에서 제외된다. 따라서, 두 경우 모두 변수 노드가 없어지므로, 사실 1은 참이다. □

무엇이 같은 그래프인가를 정의하는 것은 프로그램 표절 알고리즘을 결정한다. 정확하게 같은 모양을 가진 두 그래프를 그래프의 일치로 정의하는 대신, 두 그래프의 상대되는 노드가 같은 정보를 가지고 있을 때 두 그래프가 일치한다고 정의한다. 이 유연한 정의는 정확하게 같은 모양을 일치로 정의하는 것보다 더 많은 그래프를 일치한다고 결정한다.

**정의 4. (그래프의 일치~)** 두 프로그램 $e_1^l$와 $e_2^{l'}$에 대해서 관계식 $R(e_1^l) \sim R(e_2^{l'})$는 노드 간의 관계 $rel$을 아래와 같이 만들 수 있는 것과 필요 충분 관계를 갖는다:

$$\left( \begin{array}{l} s\ rel\ s' \Rightarrow \\ \forall (s \to n) \in R(e_1^l).\exists (s' \to n') \in R(e_2^{l'}).\ n\ rel\ n' \end{array} \right) \wedge \left( \begin{array}{l} s\ rel\ s' \Rightarrow \\ \forall (s' \to n') \in R(e_2^{l'}).\exists (s \to n) \in R(e_1^l).\ n\ rel\ n' \end{array} \right)$$

단, $rel$은 다음과 같이 정의 된다:

$$\begin{array}{lll} & s\ rel\ s' & \\ \Rightarrow & \vee & (s, s' \in Labels \wedge s = s') \\ & \vee & (s, s' \in Variable \wedge s = s') \\ & \vee & (s, s' \in DBIndex \wedge s = s') \\ & \vee & (s = \mathtt{C}(v) \wedge s' = \mathtt{C}(v') \wedge v = v') \\ & \vee & (s = \lambda(s)_0 \wedge s' = \lambda(s'_0) \wedge s_0\ rel\ s'_0) \\ & \vee & \left( \begin{array}{l} s = @(s_1, s_2) \wedge s' = @(s'_1, s'_2) \\ \wedge s_1\ rel\ s'_1 \wedge s_2\ rel\ s'_2 \end{array} \right) \\ & \vee & \left( \begin{array}{l} s = +(s_1, s_2) \wedge s' = +(s'_1, s'_2) \\ \wedge ((s_1\ rel\ s'_1 \wedge s_2\ rel\ s'_2) \vee (s_1\ rel\ s'_2 \wedge s_2\ rel\ s'_1)) \end{array} \right) \end{array}$$

□

1) 이러한 부분 그래프 생성이 모든 종류의 죽은 코드를 없애주는 것은 아니다.

정의 4에 의하면, 모양이 다른 두 그래프라도 그래프의 일치로 정의될 수 있다. 이와 같이 확장된 일치의 개념은 더 많은 프로그램 표절들을 찾아낼 수 있도록 한다. 예를 들어, 그림 8의 두 그래프는 서로 다른 모양이지만, 정의 4에 의하면 일치하는 그래프들이다.

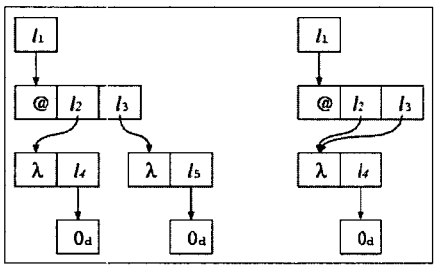


그림 8 모양은 다르지만 일치하는 그래프들

## 6. 표절 감별의 정확성

이 장에서는 4장에서 정의된 직관적인 표절 관계와 5장의 5.4절에서 제안한 그래프를 통한 표절 감별이 정확하게 일치함을 보인다. 두 개념이 정확하게 일치한다는 것은 두 개념 사이에 완전(completeness)하고 안전(soundness)한 관계가 있음을 말한다. 먼저, 두 프로그램이 정의된 표절 관계에 있을 때, 각 프로그램의 표절 감별 그래프도 일치 관계에 있음을 의미하는 완전성을 보인다. 그리고 표절 감별 그래프가 일치하는 두 프로그램은 서로 표절 관계에 있음을 의미하는 안전성을 보인다.

최종적으로 완전성 정리 1과 안전성 정리 2를 통해서 그래프를 통한 표절 감별의 정확성을 얻을 수 있다.

**정리 1. (완전성)** $P \simeq P' \Rightarrow R(P) \sim R(P')$.

증명. 증명은 $\simeq$의 개수에 관한 귀납 추론을 이용한다. 표절 관계 정의에 $\simeq$의 개수를 붙이면 아래와 같다.

$$(taut)_a \qquad e \simeq^1 e$$

$$(com) \qquad e_1 + e_2 \simeq^1 e_2 + e_1$$

$$(fix)_a \qquad \frac{g \notin fv(e)}{\texttt{fix}\ f\ \lambda e \simeq^1 \texttt{fix}\ g\ \lambda e[g/f]}$$

$$(fixrm)_a \qquad \frac{f \notin fv(e)}{\texttt{fix}\ f\ \lambda e \simeq^1 \lambda e}$$

$$(beta)_a \qquad \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} \simeq^1 e_2[e_1/x]$$

$$(tr)_a \qquad \frac{e_1 \simeq^m e_2 \quad e_2 \simeq^n e_3}{e_1 \simeq^{m+n} e_3}$$

$$(context)_a \qquad \frac{e_1 \simeq^n e_2}{C[e_1] \simeq^{n+1} C[e_2]}$$

위의 표절 관계 정의를 바탕으로

$$P \simeq^k P' \Rightarrow R(P) \sim R(P')$$

가 $k$에 관한 귀납 추론으로 증명된다. □

일반적으로 임의의 표절 감별 그래프에 대해 연관된 프로그램이 항상 존재하지는 않는다. 우리는 안전성을 위해서 임의의 표절 감별 그래프를 고려하지 않고, 제한된 형태의 표절 감별 그래프만을 가정한다. 제한된 형태의 표절 감별 그래프는 존재하는 프로그램으로부터 만들어진 그래프이다. 안전성은 두 프로그램으로부터 만들어진 표절 감별 그래프가 만일 일치한다면, 두 프로그램은 표절 관계에 있다는 정리이다. 그래프의 일치를 이용하여 프로그램의 표절로 증명을 하는 과정을 좀 더 매끄럽게 하기 위해서 우리는 다음과 같이 몇 가지 정의와 보조 정리를 사용한다.

정의 5의 역 변환 함수는 표절 감별 그래프로부터 프로그램식을 단순 명료하게 만들어 내는 함수이다. 보조 정리 1에서는 역 변환 함수에 의해 생성된 프로그램식은 표절 감별 그래프의 일치에 대해 안전함을 보이고 있다.

**정의 5.** (역 변환 함수 $T^{-1}$)

$\mathcal{T}^{-1} : Graph \rightarrow Expr$

$\mathcal{T}^{-1}[\![G|_l]\!] =$

*For* $\{l \rightarrow n\} \in G$

*case* $n$ *of*

| | | |
|---|---|---|
| $x$ | *then* | $x^l$ |
| $d$ | *then* | $d^l$ |
| $\mathtt{C}(n)$ | *then* | $n^l$ |
| $\lambda(l_0)$ | *then* | *if* $\lambda(l_0) \notin node(\|G\|_{l_0})$ |
| | | *then* $(\lambda \mathcal{T}^{-1}[\![\|G\|_{l_0}]\!])^l$ |
| | | *else* $(\texttt{fix}\ f\ \lambda \mathcal{T}^{-1}[\![\|G\|_{l_0}[f/\lambda(l_0)]]\!])^l$ |
| $@(l_1, l_2)$ | *then* | $((\mathcal{T}^{-1}[\![\|G\|_{l_1}]\!])\ (\mathcal{T}^{-1}[\![\|G\|_{l_2}]\!]))^l$ |
| $+(l_1, l_2)$ | *then* | $((\mathcal{T}^{-1}[\![\|G\|_{l_1}]\!]) + (\mathcal{T}^{-1}[\![\|G\|_{l_2}]\!]))^l$ |

단, $f \notin var(|G|_{l_0})$. □

**보조 정리 1.** $R(P) \sim R(P') \Rightarrow \mathcal{T}^{-1}[\![R(P)]\!] \simeq \mathcal{T}^{-1}[\![R(P')]\!]$.

**증명.** 이 보조 정리의 증명은 닮을 수 있는 부분 그래프에 대한 귀납 추론으로 이루어진다. 이 증명에서 중요하게 사용되는 것은 다음의 두 사실이다. 두 프로그램 $P$ 과 $P'$ 를 각각 $e^l$과 $e^{l'}$이라 할 때,

(1) 어떤 $rel$ 에 대해 $\sim$의 정의에 의해서 다음이 성립한다.

$R(P) \sim R(P') \Rightarrow l\ rel\ l' \Rightarrow (l \rightarrow n) \in R(P).\ (l' \rightarrow n') \in R(P').\ n\ rel\ n'.$

(2) 레이블 $l$ 과 $l'$에서 나가는 연결선으로 연결된 노드가 다른 레이블 $l_0$과 $l'_0$을 포함한다면, 그런 모든 레이블 $l_0$과 $l'_0$에 대하여 다음이 성립한다(예를 들어, $\{l \rightarrow \lambda l_0\} \in R(P)$, $\{l' \rightarrow \lambda l'_0\} \in R(P')$의 경우).

$$R(P) \sim R(P') \Rightarrow |R(P)|_{l_0} \sim |R(P')|_{l'_0}.$$

이 두 가지 사실과 귀납 추론을 이용하여, 보조 정리가 증명된다. □

마지막으로 역 변환 함수를 통해서 프로그램으로 변환된 프로그램식이 원래 프로그램식과 표절 관계에 있다는 것을 보일 필요가 있다. 이 과정을 위해서 중간 단계의 프로그램 형태인 끼운(inlined) 프로그램을 정의하고, 본래 프로그램의 끼운 프로그램이 역 변환으로 나온 프로그램식과 표절 관계에 있음을 보조 정리 2에서 보인다. 정의된 표절 감별 그래프에서는 **let** 프로그램식을 나타낼 방법이 없으므로, **let** 프로그램식에 대해서는 귀납적으로 증명할 수가 없기 때문에 증명을 위해서 **let**을 없앤 끼운 프로그램식을 다시 정의한 것이다. 또한, 기존의 프로그램식을 끼운 프로그램식으로 바꾸는 것이 프로그램의 의미를 해치지 않으므로 이같은 중간 단계의 삽입은 문제될 것이 없다. 최종적인 목표인 정리 2를 위해서 마지막 보조 정리인 보조 정리 3에서는 본래의 프로그램과 프로그램의 표절 감별 그래프로부터 역 변환 함수를 통해 생성된 새로운 프로그램이 표절 관계에 있음을 보인다.

**정의 6. (끼운 프로그램 $P_I$)** 어떤 프로그램 $P$의 끼운 프로그램 $P_I$는 아래와 같이 정의된다(정의에서는 $P$ 대신 프로그램식 $e$를 사용한다.):

$$
\begin{array}{lll}
e_I \triangleq \textit{Case } e \textit{ of} & & \\
\quad d & \textit{then} & d \\
\quad v & \textit{then} & v \\
\quad x & \textit{then} & x \\
\quad \lambda e_1 & \textit{then} & \lambda(e_1)_I \\
\quad \texttt{fix } f\ \lambda e_1 & \textit{then} & \texttt{fix } f\ \lambda(e_1)_I \\
\quad e_1\ e_2 & \textit{then} & (e_1)_I\ (e_2)_I \\
\quad e_1 + e_2 & \textit{then} & (e_1)_I + (e_2)_I \\
\quad \texttt{let } x = e_1 \texttt{ in } e_2 \texttt{ end} & \textit{then} & (e_2)_I[(e_1)_I/x]
\end{array}
$$

**보조 정리 2.** $\mathcal{T}^{-1}[\![R(P)]\!] \simeq P_I$. 단, $P_I$는 $P$의 끼운 프로그램이고, $\mathcal{T}^{-1}[\![R(P_I)]\!]$와 $P_I$에 나타나는 변수 이름은 전역 변수를 제외하고는 다르다고 가정한다.

**증명.** 프로그램식에 대한 귀납적 추론으로 증명할 수 있다. 여기서 사용하는 프로그램식은 **let** 프로그램식을 없앤 프로그램식이므로, 귀납적 방법론으로 증명된다. □

**보조 정리 3.** $\mathcal{T}^{-1}[\![R(P)]\!] \simeq P$.

**증명.** 이 보조 정리의 증명은 아래의 네 가지 사실로부터 보일 수 있다.

(1) $P \simeq P_I$: 이 사실은 정의 6에서부터 어렵지 않게 증명된다. 왜냐하면, 정의 6에서 사용하는 "*then*"은 $\simeq$ 정의를 만족하고, 정의 6의 합성적(compositional) 정의 방법은 표절 관계의 (context)와 (tr) 규칙을 만족하기 때문이다.

(2) 정리 1에 의해서, (1)로부터 $R(P) \sim R(P_I)$이 성립한다.

(3) 보조 정리 1에 의해서, (2)로부터 $\mathcal{T}^{-1}[\![R(P)]\!] \simeq \mathcal{T}^{-1}[\![R(P_I)]\!]$이 성립한다.

(4) 보조 정리 2에 의해서, $\mathcal{T}^{-1}[\![R(P_I)]\!] \simeq P_I$이 성립한다.

따라서 표절 관계의 (context) 규칙과 (1), (3), (4)에 의해서 $\mathcal{T}^{-1}[\![R(P)]\!] \simeq P$이 성립한다. □

**정리 2. (안전성)** $R(P) \sim R(P') \Rightarrow P \simeq P'$.

**증명.** $R(P) \sim R(P')$

$\Rightarrow \mathcal{T}^{-1}[\![R(P)]\!] \simeq \mathcal{T}^{-1}[\![R(P')]\!]$ 보조 정리 1에 의하여

$\Rightarrow P \simeq P'$ 보조 정리 3과 표절 관계의 (tr) 규칙에 의하여 □

정리 2는 표절 감별 그래프를 이용한 프로그램 표절 감별이 올바름을 말하고 있다. 정리 2에 의해서, 표절 감별 그래프가 일치하는 두 프로그램은 표절된 프로그램이라고 말할 수 있다.

## 7. 실 험

우리는 제안된 방법론을 실제로 사용되는 프로그래밍 언어인 nML[10]에 적용하여 구현하였다. 우선, 두 개의 프로그램이 앞서 정의된 표절 관계에 있음을 표절 감별 그래프를 이용한 방법으로 구현하였다. 구현된 프로그램을 이용하면 두 프로그램이 표절 관계에 있는지 없는지 손쉽게 알 수가 있다.

하지만, 일반적으로 두 프로그램이 정확하게 일치하는 표절 감별 그래프를 가지게 되는 경우는 많지 않다. 따라서, 프로그램 표절 감별의 폭을 넓히기 위해서, 두 프로그램의 일부가 표절 관계에 있는 경우를 포괄하는 비교 알고리즘이 표절 감별 검사 도구에 포함될 필요가 있다. 기존의 여러 가지 구조적인 비교 알고리즘 중에서, 우리는 CloneChecker의 알고리즘을 사용하여 프로그램의 일부가 표절 관계인 경우를 고려하였다.

CloneChecker의 알고리즘에서 정의되는 프로그램의 유사성은 다음과 같은 식으로 나타난다.

$$\text{유사도}(a, b) = \frac{S_a + S_b}{T_a + T_b}$$

여기서, $a$와 $b$는 트리 구조로 변환된 프로그램이고, $S_a$는 $b$에 나타나는 $a$의 부분 트리의 개수이다. 마찬가지로 $S_b$는 $a$에 나타나는 $b$의 부분 트리의 개수이다. $T_a$와 $T_b$는 각각 $a$와 $b$의 부분 트리의 개수를 의미한다. 유사도는 0과 1 사이의 숫자로 나타나는데, 0에 가까워질수록 두 프로그램이 비슷하지 않음을, 1에 가까워질수록 비슷함을 의미한다.

다음의 nML 예제를 살펴보자.

**예제 3.** 다음은 프로그램 표절의 예제이다.

```
let val a = let fun f x = x
                    val y = 1
                in
                    (f y) + y
                end
in
    a                                  과
end
```

```
let val dlst = [1,2,3,4,5,6]
    fun dfun x = List.mem x dlst
    val c = 1
    val d = c + ((fn x=>x) c)
in
    d
end
```

위 두 프로그램은 결과를 정수 2로 내는 동일한 일을 하는 프로그램이다. 왼쪽의 프로그램에서는 함수 **f**와 변수 **y**의 값을 **let**을 이용해서 정의하여 사용한다. 오른쪽의 프로그램에서는 함수 **f**를 사용하지 않고, 직접 해당하는 함수 정의를 이용하였다. 또한 오른쪽의 프로그램은 표절하였음을 속이기 위한 수행되지 않는 코드 **dlst**와 **dfun**의 정의가 삽입되었다. □

CloneChecker의 문법 구조 트리를 바탕으로 하는 검사를 이용해서 예제 3의 프로그램을 검사하면 0.105의 유사성이 나오는 반면, 제안된 표절 감별 그래프를 이용한 검사 방법을 사용하면 1.0의 유사성이 나온다. 그 이유는 예제 3의 프로그램들이 4절에서 정의한 표절 관계를 가지기 때문이다.

일반적으로 논문에서 제안하는 방법에 의한 유사도가 CloneChecker에 의한 것보다 항상 분별력 있는 결과를 주는 것은 아니다. 제안된 방법은 프로그램 문법 정보를 내부적으로 변환하여 사용하기 때문에, 문법적으로 유사하지만 수행 흐름이 다른 프로그램들에 대해서는 CloneChecker에 비해 높은 유사도를 주지 못한다. 또한, 제안된 방법에서 **let** 바인딩을 제거하는 대신 프로그램식을 복사하는 효과를 가지므로, **let**에 바인딩된 프로그램식만 다른 두 프로그램을 검사하는 경우에는 CloneChecker 보다 못 미치는 표절 감별율을 가질 수도 있다.

그림 9는 2003년도 학부과정 프로그래밍 언어의 과제로 제출된 프로그램들을 우리의 검사 도구와 CloneChecker로 검사한 결과이다. 이 과목은 원래 90명의 학생들이 수강하는 과목이어서 자동 표절 검사 도구 없이 표절을 찾아내기 힘든 과목이다. 그림 9의 표는 60개의 제출된 과제 중에서 10개의 샘플을 뽑아서 그들간의 유사도를 나타낸 것이다. 표의 가로축은 비교가 수행된 프로그램짝이고 표의 세로축은 각 프로그램짝의 유사도를 나타낸 것이다. 그래프는 CloneChecker의 결과로부터 유사도가 낮은 것에서 높은 것의 순으로 그려졌다.

각각의 검사 도구에서 프로그램 구조를 단순화하는 정도나 비교하는 방법이 구현에 따라 매우 다르기 때문에 두 도구에서 나온 유사도를 함께 놓고 비교할 수는 없다. 하지만, Clonechecker에 대해선 0.9의 유사도를 기준으로, 우리의 방법에선 0.8의 유사도를 기준으로 프로그램짝이 표절 되었을 확률이 매우 높음을 알 수 있다.

그림 9의 결과로 부터, 우리는 오른쪽 세 개의 프로그램짝들이 표절의 가능성이 있음을 알아낼 수 있다. 실제로 검사 당시에는 CloneChecker 만 구현이 된 상태였고, CloneChecker에서 0.9 이상의 유사도를 보인 두 개의 프로그램짝들 만을 표절로 의심, 학생들과 면담을 수행했었다.

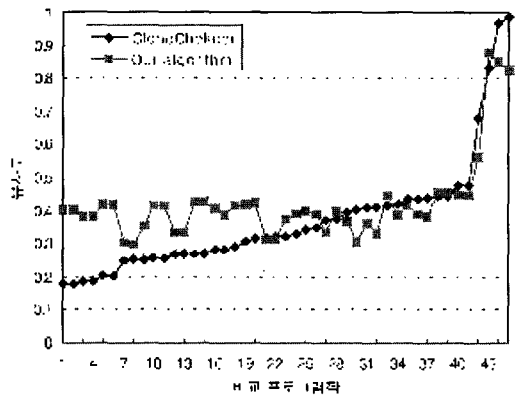


그림 9 제안된 방법과 CloneChecker로 비교된 프로그램 그룹들

CloneChecker 에서 비교적 낮은 유사도를 보였던 한 프로그램짝(CloneChecker에서의 유사도 0.83)은 우리의 방법에서 가장 높은 유사도를 나타내었다. 저자가 그 프로그램짝을 직접 눈으로 비교해 본 결과, 예상과 같이 let 등의 코드를 이용하여, CloneChecker가 감별하지 못할 정도의 작은 수정을 시도한 프로그램짝이었다.

실험 데이타를 통해서, 구현된 검사 도구가 기존의 CloneChecker와 상호 보완적으로 프로그램 표절 검사에 사용될 수 있음을 확인할 수 있다. 또 더 나아가서, 우리의 표절 감별 방법에 걸맞는 가중치 있는 부분 프로그램 비교 알고리즘을 고안한다면, 지금보다 높은 분별력을 가진 표절 검사 도구가 될 수 있을 것이다.

## 8. 결론 및 앞으로 할 일

이 연구에서는 프로그램 표절 감별을 위해서 프로그램의 표절 감별 그래프를 제시하였다. 프로그램을 제시된 그래프로 바꾸어 비교를 하면, 프로그램의 문법 정보뿐 아니라, 바인딩 정보까지도 드러낼 수 있기 때문에, 문법 구조만을 비교하는 이전의 방법보다 훨씬 더 효과적으로 표절 프로그램을 찾아낼 수 있다. 우리는 또한 표절 프로그램이란 무엇인가를 엄밀하게 정의하고 이 표절 프로그램의 정의와 표절 감별 그래프와의 관계를 보였다. 즉, 두 프로그램이 표절이라는 것은 표절 감별 그래프가 일치한다는 것과 필요 충분 관계에 있음을 증

명하였다.

1장에서 요약하였던 표절 시도 중에서, 제안된 방법론은 다음과 같은 표절 시도를 막을 수 있다: (1) 변수의 유효범위를 조작하는 것은 표절 그래프에 드러난 바인딩 관계에 의해 해결되고, (2) 변수들의 이름을 바꾸는 것은 De Bruijn 표기법을 통해서 막을 수 있고, (3) 교환 법칙이 가능한 연산의 피연산자를 바꾸는 시도는 표절 그래프 일치의 정의에서 포함되고, (4) 죽은 코드를 삽입하는 시도는 표절 그래프 정의 중 닿을 수 있는 부분 그래프 생성을 통해서 일부를 막을 수 있다. 하지만, 제안된 방법론도 역시 프로그램의 의미보다는 문법 구조에 치우쳐 비교를 하므로, (5) 부분적인 수행 (예를 들어, 상수로 바꾸기) 에 관해서는 표절 시도를 찾아 낼 수 없는 단점이 있다. 제안된 방법의 가장 큰 장점 중 하나는 (1)에서 (4)의 일부를 포함하는 표절 시도를 한꺼번에 여러 번 적용하여도 완벽하게 표절 감별을 할 수 있다는 것이다.

제안된 방법의 구현을 통해서, 기존의 nML 프로그램 표절 감별에 본 연구가 기존의 연구와 함께 효율적으로 사용될 수 있음으로 확인하였다. 기존의 문법 구조 기반의 표절 감별 연구에서 찾아내지 못하는 프로그램 표절을, 제안된 방법론은 찾아낼 수 있다. 결론적으로, 제안된 방법론이 기존의 문법 구조 기반의 표절 감별 연구에서 다루지 못하는 흐름 구조를 이용한 표절의 시도를 확실하게 감별함으로써, 기존의 방법과 함께 다양한 표절 시도를 막을 수 있는 보다 강력한 표절 감별 도구를 개발하였다는 데에 의의가 있다.

## 참 고 문 헌


[1] 장성순, 서선애, 이광근, "프로그램 유사성 검증기", 한국정보과학회, 가을 학술 회의, 제 28권, pp. 334-336, 2001년 10월.

[2] 서선애, 한태숙, "흐름 그래프 형태를 이용한 프로그램 유사성 비교", 학술 논문, 한국과학기술원 전산학과 프로그래밍 언어 연구, 2004년 10월. http://ropas.kaist.ac.kr/~saseo/papers/kor_simil_long.pdf .

[3] Dick Grune, "SIM: The software and text similarity tester SIM," http://www.few.vu.nl/~dick/sim.html.

[4] M. J. Wise, "YAP3: improved detection of similarities in computer programs and other texts," SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education)," volume 28, 1996.

[5] Alex Aiken, "Moss: a system for detecting software plagiarism," http://ftp.cs.berkeley.edu/~aiken/moss.html.

[6] Guido Malpohl, "JPlag: Detecting Software Plagiarism," http://wwwipd.ira.uka.de:2222/.

[7] 황미녕, 강은미, 조환규, "유전체 서열의 정렬 기법을 이용한 소스 코드 표절 검사", 한국정보과학회, 2002년 제 21회 논문경진대회 최우수상, 2002년 6월, http://jade.cs.pusan.ac.kr/bioinformatics/bio.html.

[8] 이평준, 전명재, 조환규, "재귀적 지역정렬을 이용한 프로그램 표절 탐색", 한국정보과학회 봄 학술 회의, 2004년 4월.

[9] N. De Bruijn, "Lambda-calculus Notation with Nameless Manipulation," A Toolfor Automatic Formula Manipulation, volume 34, pp. 381-392, 1972.

[10] 이광근, 이욱세, 어현준, 김정택, 최웅식, 류석영, 강현구, 서선애, 장성순, 김범식, "nML 컴파일러 시스템 (status report)", 한국정보과학회 가을 학술 회의, 제 28권, pp. 340-342, 2001년 10월. http://ropas.kaist.ac.kr/n



서 선 애

1994년~1998년 한국과학기술원 전산학과(학사). 1998년~2000년 한국과학기술원 전자전산학과 전산학전공(석사). 2000년~현재 한국과학기술원 전자전산학과 전산학전공 박사 과정. 관심분야는 프로그래밍 언어, 프로그램 분석, 컴파일러



한 태 숙

1972년~1976년 서울대학교 전자공학과(학사). 1976년~1978년 한국과학기술원 전산학과(석사). 1990년~1995년 Univ. of North Carolina at Chapel Hill(박사). 1996년~현재 한국과학기술원 전자전산학과 교수. 관심분야는 프로그래밍 언어론, 함수형 언어, 임베디드 시스템 설계 및 분석