선택적 레덱스 트레일 기반의 디버거
(A Debugger based on Selective Redex Trail)

박 희 완 † 한 태 숙 ††
(Heewan Park) (Taisook Han)

요 약 함수형 프로그래밍 언어는 전통적인 프로시저형 언어에 비하여 많은 장점이 있다. 그러나 함수언어 프로그래머를 위한 실용적인 디버깅 환경은 상대적으로 빈약하다.

그동안 유용한 디버거 구현을 위해서 많은 시도가 있었고, 그 결과로 하향식 기법을 이용한 알고리즈믹 디버거와 상향식 기법을 이용한 레덱스 트레일 디버거가 연구되었다. 두가지 기법은 모두 실제 프로그래밍에 적용하기에는 유지해야 하는 디버깅 정보의 양이 많다는 단점이 있다.

이 논문에서는 선택적 레덱스 트레일 디버깅 방법을 제안한다. 이 방법을 이용하면 디버거 사용자는 프로그램에서 오류가 예상되는 부분에 포커스를 설정할 수 있고 단지 선택된 부분에 한하여 트레일을 생성하게 된다. 이 방법은 프로그램의 오류에 대한 디버거 사용자의 예측을 반영하고 디버깅에 필요한 정보의 양을 줄이는 장점이 있다. 구현된 디버깅 시스템은 선택적 레덱스 트레일을 생성하는 추상기계와 실제 디버깅이 이루어지는 레덱스 트레일 탐색기로 구성된다.

Abstract Information Feature Visualization integrates the traditional information retrieval techniques and visualization techniques and reduces the time and effort from searching for requested information by promoting apprehension from the wide and various internet information. It's used to understand the related information tendency by supporting the statistic materials for retrieved results. It's useful to catch a tendency regarding information over the search process. Recently, by increasing interest in the information visualization in the library system, The research about various visualization technology and searching system has been developed by many researchers. So, to describe systematically and visually the feature of information, we'll make the element of information visualization uniform by various views and suggest paradigm renewing retrieval technology using information outlining techniques by developing search tool to improve apprehension about information and navigate more easily among information. We can improve user's apprehension about search information and search more usefully.

1. 서 론

함수형 프로그래밍 언어는 간결하고 구조적이며 부작용이 없고(side-effect free) 프로그램을 쉽게 모듈화 할 수 있기 때문에 명령형 언어(imperative language)에 비하여 프로그램 생산성이 높다는 장점을 가지고 있다[1]. 그럼에도 불구하고 함수형 프로그래밍 언어는 명령형 언어와 비교하면 상대적으로 사용자가 적은데, 그 이유 중 하나는 빈약한 디버깅 환경에 있다. 명령형 언어를 이용하는 프로그래머에게는 선택할 수 있는 많은 디버깅 도구들이 존재하는 반면, 함수형 언어 프로그래밍에 적합한 실제적이고 범용적인 디버깅 도구가 부족하다[2,3,4].

엄격한 타입을 가지는 함수형 언어에서는 대부분의 에러들이 프로그램이 실행되기 전에 타입검사 시스템에 의해서 발견될 수 있기 때문에 디버거의 필요성에 대한 논란이 있었다[5,6]. 타입검사 시스템이 프로그램이 실행되기 전에 많은 에러를 찾아낼 수 있는 것은 사실이지만 프로그램 내에 존재하는 모든 에러들을 찾을 수 있는 것은 아니다. 프로그램 수행 도중 발생하는 실행시간 오류(runtime error)나 논리적인 오류를 찾기 위해서

· 본 논문은 첨단정보기술연구센터를 통하여 과학재단의 지원을 받았음.
† 학생회원 : 한국과학기술원 전자전산학과
hwpark@pllab.kaist.ac.kr
†† 종신회원 : 한국과학기술원 전자전산학과 교수
han@cs.kaist.ac.kr
논문접수 : 1999년 4월 2일
심사완료 : 2000년 7월 8일

디버거가 유용하게 사용될 수 있다[7].

명령형 프로그래밍 언어에서 유용하게 쓰이고 있는 전통적인 디버깅 기법으로 프로그램 내에 프린트(print) 명령을 삽입하여 프로그래머가 원하는 정보를 얻어내는 방법이 있지만, 지연 함수형 언어로 작성한 프로그램은 부분적으로 필요할 때만 계산(evaluation)되는 무한 자료구조와 완전하게 평가되지 않은 표현식(expression)이 존재할 수 있다는 문제가 있다.

함수형 언어에서 이용되는 대표적인 디버깅 기법중에는 하향식(top-down)방법으로 오류의 위치를 찾아내는 알고리즈믹 디버깅(algorithmic debuggging)기법[8]과, 상향식(bottom-up)방법으로 오류의 위치를 찾아내는 레덱스 트레일 디버깅(redex trail debugging)기법[4]이 있다.

알고리즈믹 디버깅 기법은 디버깅 시스템이 요구하는 질문에 대한 디버거 사용자의 정확한 답변을 전제로 오류의 위치를 정확하게 알아낼 수 있는 장점이 있다. 하지만 디버깅에 이용되는 EDT(evaluation dependence tree)의 크기에 대한 부담과 디버깅 시스템으로부터의 질문의 수가 많다는 단점이 있다. 이 단점을 보완하고자 부분적으로 생성된 EDT에 대한 디버깅을 시도하고 나머지 부분에 대해서 다시 EDT의 생성과 디버깅을 반복하는 방법이 제안되었지만 디버깅 시스템으로부터의 질문의 수에는 변함이 없고, 단지 여러 단계로 디버깅을 나누어 시도하는 의미가 있다. 레덱스 트레일 디버깅 기법은 어떤 프로그램에 실행시간 오류(run-time error)가 생겼을 때 그 에러의 원인은 프로그램이 비정상적으로 실행 종료된 부근에 존재한다는 고찰에서 출발하였다. 디버거 사용자는 레덱스 트레일의 추적 단계를 직접 선택해야 하기 때문에 대상 프로그램을 충분히 이해하고 있다면 쉽게 오류의 위치를 찾아낼 수 있다. 하지만 모든 축약 단계(reduction step)마다 생성되는 레덱스 트레일의 사이즈가 크다는 단점이 있다. 이 단점을 보완하고자 정확한 동작이 보장되는 라이브러리나 모듈에 대하여 레덱스 트레일을 생성하지 않는 방법이 제시되었지만[9] 레덱스 트레일의 크기를 줄이는 근본적인 해결책이 되지는 못했다.

본 논문에서는 디버거 사용자가 전체 프로그램 중에서 디버깅을 원하는 부분에만 동적으로 포커스를 설정하여 부분적인 트레일을 생성하는 선택적 레덱스 트레일 기법을 제안한다. 이 기법은 디버거 사용자가 가지고 있는 오류의 위치에 대한 직감을 디버깅 정보로 이용할 수 있다는 장점이 있다. 그리고 포커스가 설정된 부분에 한하여 세부적인 트레일을 생성하기 때문에 디버깅에 필요한 트레일 사이즈를 줄일 수 있다. 또한, 표현식이 계산되기 전의 트레일 정보와 계산된 후의 트레일 정보를 하나의 쌍으로 묶어서 EDT를 구성할 수 있기 때문에 레덱스 트레일 디버깅 기법에서의 상향식 디버깅과 함께 알고리즈믹 디버깅 기법에서의 하향식 디버깅이 가능하다.

본 논문에서 구현한 시스템의 구체적인 디버깅 대상은 지연 계산 기반 언어의 의미식 기반으로 생성한 추상기계 언어에 제한된다. 따라서, 기존의 함수형 언어 소스 프로그램을 추상기계 언어로 바꾸어서 디버깅 테스트를 했다. 그러나 기존의 함수형 언어 프로그램을 본 논문에서 대상으로 삼은 추상기계 언어로 변환하는 변환기가 구현될 경우 디버깅 대상은 추상기계 언어로 변환될 수 있는 모든 함수형 언어로 확장될 수 있다. 또한, 본 논문에서 제안한 디버깅 아이디어는 부작용(Side Effect)이 없이, 계산(Evaluation)기반으로 동작하는 순수 함수형 언어(Pure Functional Language) 시스템에 적용될 수 있다.

기존의 디버깅 방법은 디버거 환경에 대한 이식성을 중시했기 때문에 프로그램 변환기법을 이용하였다. 본 논문에서 제안하는 선택적 레덱스 트레일 디버깅 기법은 추상기계 상에서 구현되었기 때문에 프로그램 변환기법을 기반으로 하는 디버거에 비하여 이식성이 부족하다는 단점이 있다. 그 이유는 프로그램 변환기법을 이용하는 디버거의 경우, 디버깅 대상 프로그램에 디버깅 정보(EDT 생성 루틴, EDT 순회 디버깅 루틴 포함)를 추가된 새로운 소스 프로그램을 생성시키는 것을 목적으로 하기 때문에 어떠한 시스템에서도 소스 프로그램을 컴파일하여 디버깅 할 수 있기 때문이다. 그러나 추상기계를 기반으로 하는 디버거의 경우에는 추상기계가 동작하는 시스템 환경하에서 디버깅이 가능하다는 제한이 있다. 그러나 추상기계 기반으로 동작하기 때문에 디버깅 시스템이 동작하는 도중에도 디버깅하기를 원하는 부분에 포커스를 설정할 수 있다는 특징이 있다.

본 논문의 구성은 다음과 같다. 제 2 장에서는 관련 연구로써 명령형 언어에서 쓰이는 전통적인 디버깅 기법과 함수형 언어의 특성을 고려하여 제안된 알고리즈믹 디버깅 기법과 레덱스 트레일 디버깅 기법을 설명한다. 제 3 장에서는 기존의 디버깅 기법의 단점을 보완하여, 본 논문에서 제안하는 선택적 레덱스 트레일을 이용한 디버깅 기법에 대해서 설명한다. 제 4 장에서는 실제로 구현한 추상기계와 트레일 탐색기를 설명하고, 제 5 장에서 결론 및 향후과제에 대하여 설명한다.

2. 관련 연구

이 장에서는 전통적으로 명령형 언어에서 쓰이던 전통적인 디버깅 기법들에 대해서 설명하고 이 방법들이 함수형 언어에 적용되기 어려운 원인을 알아본다. 그리고 함수형 언어의 특성을 고려한 디버깅 기법으로, 이 논문의 바탕이 된 알고리즈믹 디버깅 기법과 레덱스 트레일 디버깅 기법을 설명한다.

2.1 전통적인 디버깅 방법

함수언어를 위한 디버거를 설계하기 위해서 먼저 명령형 언어에서 쓰이던 전통적인 디버깅 기법들을 고려할 수 있다. 프로그램 중간 중간에 프린트(print)명령을 삽입하는 방법, 단계적 실행(single-stepping), 변수값 검사방법(variable watching) 등이 전통적인 명령형 언어 디버깅 환경에서 쓰이는 방법이다. 명령형 언어에서 이러한 기법들이 유용한 이유는 프로그램 실행 순서와 프로그램 소스와의 밀접한 관계가 보장되고, 표현식(expression)의 값이 평가되지 않은 채로 남아있는 경우가 없기 때문이다.

그러나 함수형 언어에서 이러한 방법을 그대로 적용하는 것은 유용하지 못하거나 이용 자체가 불가능한 경우도 있다[7,10]. 지연함수 언어는 프로그램의 실행순서를 프로그램 소스만으로 예측하기가 어렵고 값 형태로 계산(evaluation)되지 않고 남아있는 표현식(expression)을 검사하는 것은 무의미할 수도 있기 때문이다.

전통적인 디버깅 기법이 함수형 언어에 적용된 예로는 Haskell의 trace함수와 show함수를 들 수 있다.

```
trace :: String -> a -> a
```

trace함수는 입력 인자중에서 두번째 값을 반환하는데 그와 함께 부가적인 효과(side-effect)로써 첫번째 인자를 출력하게 된다. 하지만 이 방법은 프로그램에서 실제로 계산(evaluation)되지 않는 표현식에 대해서도 계산을 강행하기 때문에 프로그램의 의미 자체가 달라지는 경우도 발생할 수 있다. 그리고 프로그램상에서 필요로 할 때만 부분적으로 이용될 수 있는 무한 자료구조에 trace함수를 이용할 경우에는 정상적으로 종료 가능한 프로그램이 종료되지 않는 프로그램으로 바뀔 수도 있다[3,7].

따라서 함수형 언어의 특성을 충분히 고려한 디버깅 기법이 필요하게 되었고, 대표적인 기법으로 이 논문의 바탕이 된 알고리즈믹 디버깅 기법과 레덱스 트레일 디버깅 기법을 살펴본다.

2.2 알고리즈믹 디버깅(Algorithmic Debugging)

Shapiro[8]에 의해 제안된 알고리즘 디버깅은 프로그램 디버깅에 이론적인 기반을 갖게 하는 첫 번째 시도였고, 자동 디버거(automatic debugger)의 부분적인 기초가 되었다.

알고리즈믹 디버깅에서는 대상 프로그램의 원하지 않는 결과가 나오게 된 원인을 찾기 위해서 프로시저의 호출 관계를 트리로 만들어 가장 상위레벨의 노드에서부터 탐색을 시작한다. 이러한 트리를 EDT라고 하는데, EDT(Evaluation Dependence Tree)는 함수형 언어에서 각각의 함수에 의해서 계산(Evaluation)된 값과, 그 값을 얻게 된 원인이 되는 함수 정보를 동시에 유지하는 트리 구조를 폭넓게 지칭한다.

만약 프로그램에 오류가 있다면 그 오류는 가장 상위레벨의 프로시저에 존재하거나 상위레벨의 프로시저로부터 호출되는 그밖의 다른 프로시저에 존재한다. 최상위 레벨로 부터 호출되는 모든 하위 프로시저가 정상적으로 동작한다는 것이 확인되면 최상위 레벨의 프로시저에 에러의 원인이 있고, 그렇지 않으면 또 다른 하위 프로시저의 호출 관계에서 오류의 원인을 찾을 수 있게 되는 것이다. 하지만 디버거는 정상적인 동작과 버그 증상을 구분할 수 없기 때문에 적어도 프로그램의 동작을 정확히 판단할수 있는 전지자(oracle)가 존재하여 이러한 정보를 제공해 주어야 한다. 일반적으로 프로그램 디버깅에 임하는 디버거 사용자가 이 역할을 하게 된다.

알고리즈믹 디버깅은 프로그램 동작의 정확성에 대한 질문을 하여 디버거 사용자의 답변을 얻고, 이를 기반으로 오류가 존재하는 어떤 프로시저나 함수를 찾아낸다. 그러나 알고리즈믹 디버깅은 한번의 디버깅에 하나의 오류만을 찾아낼 수 있기 때문에 프로그램에 하나 이상의 오류가 존재하는 경우에는 알고리즈믹 디버깅을 반복해서 적용하여 다른 오류를 찾아내어야 한다.

```
start from  EDT->root-node;
error-node-find-flag = FALSE;
do {
    printf "Is it correct ?" : current-node->equation;
    get user's answer;
    if ( user's answer is "Yes" ) {
        next-node = current-node -> sibling-node;
        if ( next-node is empty ) {
            error-node = current-node->parent;
            error-node-find-flag = TRUE;
        }
    }
    else /* user's answer is "No" */ {
        next-node = current-node -> child-node;
```

```
        if ( next-node is empty ) {
            error node = current-node;
            error node-find-flag = TRUE;
        }
    }
    current-node = next-node;
} while( error-node-find-flag is not TRUE );
print "Error is located in " : error node -> name;
```

그림 1 알고리즈믹 디버깅 기법에서 이용하는 디버깅 알고리즘

알고리즈믹 디버깅이 이루어지는 과정을 살펴보면 다음과 같다. 디버거는 먼저 프로그램을 수행하고 프로시저 레벨에서 실행 의존 트리(EDT:execution dependence tree)를 만든다. 트리의 노드는 각각의 프로시저 호출에 의해서 생성되고, 프로시저 이름과 모든 함수 입출력 인자와 같은 필수적인 추적(trace)정보가 저장된다. 만약 현재 노드로부터 호출되는 또 다른 프로시저가 있을 경우에는 이 프로시저는 현재 노드의 자식 노드(child node)가 된다. 새로운 자식 노드들은 프로시저 호출 순서에 따라서 왼쪽에서 오른쪽으로 현재 노드에 삽입이 된다. 일단 실행이 끝나면 알고리즈믹 디버거는 EDT 트리를 순회하면서 버그를 찾기 시작한다. 버그를 찾는 알고리즘은 그림 1과 같다. 디버거 시스템은 각각의 트리 노드에 대하여 프로시저 호출에 따라 노드의 행동이 올바른지의 여부를 디버거 사용자에게 물어보고 사용자는 이러한 질문에 '예' 혹은 '아니오' 라고 대답한다. 만약 '예'라는 답변이 주어진다면 현재 노드의 서브 트리들이 정상적으로 동작한다는 것을 의미하므로 트리탐색은 현재 노드의 형제(sibling) 노드에서 탐색이 다시 이루어진다. 만일 '아니오'라는 답변이 주어진다면 현재 노드의 서브 트리들 중에서 오류의 원인이 존재하는 것을 의미하므로 현재 노드의 가장 왼쪽 자식 노드에서 트리 탐색을 계속한다. 이 탐색은 현재 노드에서 '예' 라는 응답을 얻었으나 형제 노드가 존재하지 않는 경우이거나, '아니오' 라는 응답을 얻었으나 더 이상 탐색을 계속 할 자식 노드가 없는 경우에 종료되며 이때 디버거는 잘못된 프로시저의 이름을 알려주게 된다.

2.3 알고리즈믹 디버깅 기법의 평가

알고리즈믹 디버깅 기법은 하향식(top-down)으로 동작한다. 프로그램의 메인 함수(main function)에서 시작하여 그 함수에서 호출하는 서브 함수(sub function)들의 행동의 옳고 그름의 질문에 대한 디버거 사용자의 응답을 바탕으로 디버깅 하는 방식이다. 만약 디버깅 시스템이 요구하는 질문에 대한 디버거 사용자의 정확한 답변을 전제로 한다면 디버거는 오류의 위치를 정확하게 알아낼 수 장점을 갖고 있다[11]. 따라서 이 기법은 오류의 위치에 대한 디버거 사용자의 직감을 필요로 하지 않기 때문에 전체적인 프로그램의 이해를 갖지 않고 있더라도 충분히 디버깅 효과를 거둘 수 있다.

알고리즈믹 디버깅 기법을 실제로 사이즈가 큰 프로그램에 적용하기에는 몇가지 문제점이 있다[3,7,13]. 그 중 첫번째는 EDT의 크기 문제이다. 알고리즈믹 디버깅 기법에서는 모든 축약단계(reduction step)가 모두 EDT에 저장되는데 이것은 EDT의 사이즈가 쉽게 수백 메가 바이트까지 도달할 수 있다는 것을 의미한다[13]. 컴퓨터 성능의 발달로 메인 메모리의 크기가 EDT를 모두 수용할 수 있을 만큼 커지는 추세이지만 그럼에도 불구하고 여전히 EDT의 크기는 그 한계값(upper bound)이 고정되지 않는다는 문제점을 가지고 있다. 이에 대한 해결책으로 동작이 확실하다고 보장되는 모듈에 대해서 EDT를 만들지 않는 Thin Tracing방법[12]과 전체 EDT를 한번에 만들지 않고 부분적으로 나누어 생성하는 Piecemeal Tracing[12]방법이 제시되었다. Thin Tracing 방법으로 전체 EDT 크기를 50%정도로 줄인 사례가 있지만 프로그램의 사이즈가 커질 경우에는 디버깅을 위한 근본적인 해결책이 되지는 못한다. Piecemeal Tracing 방법을 적용할 경우에는 부분적으로 생성된 EDT 에 대한 컴파일과 디버깅을 반복해야 하는 단점이 있다.

EDT의 크기가 커짐에 따라 발생하는 또다른 문제점은 디버거 사용자가 너무 많은 질문에 응답해야 하는 것이다[7,13]. 이것은 알고리즈믹 디버깅 시스템을 간단한 디버깅에 이용하기에 매우 번거롭게 하는 원인이기도 한데, EDT 사이즈를 줄임으로써 이 문제를 간접적으로 해결할 수 있다. 하지만 지연 함수형 언어에 있어서 특징적으로 나타나는 계산되지 않은(unevaluated)형태의 표현식에 대한 질문은 답변 자체가 무척 어려울 수 있다. 이러한 계산되지 않은 표현식에 대한 질문의 형태를 보완하기 위해서 EDT를 이루고 있는 노드들을 최대한 값 형태로 변화시켜주는 기법인 EDT Strictification[12]이 제안되었다. 이 기법에서는 계산되지 않은 채로 남아 있는 표현식들을 최대한 계산된 값으로 대치하는 방법을 이용하였는데, EDT 생성 루틴에 표현식의 값을 대치시키는 과정이 포함되었기 때문에 EDT 생성시간이 상대적으로 늘어난다는 단점이 있다.

2.4 레덱스 트레일 디버깅(Redex Trail Debugging)

레덱스 트레일 디버깅 기법[4,9]은, 어떤 프로그램이 실행시간 오류를 일으켰을 때 그 오류의 원인이 일반적으로 프로그램이 비정상적으로 실행 종료된 부근에 존재하게 된다는 고찰로부터 출발한다. 따라서 디버깅의 시작점을 선정하는데 있어서 알고리즈믹 디버깅에서와 같이 프로그램의 시작을 선택하는 것보다는 비정상적으로 종료된 바로 그 위치로 결정하는 것이 더 유용하다는 것이다. 여기서 레덱스 트레일이라는 것은 결과로 나온 값이나 오류로부터 시작하여 그 원인을 제공한 부모 레덱스(parent redex)를 연결하고 또 현재 레덱스의 부모 레덱스를 연결하는 작업을 입력 데이타나 초기 표현식이 나타날 때까지 반복한 것을 의미한다.

```
foo xs | head xs < 0 = 0
       | otherwise = head xs
fie xs = ( ( foo xs ) : fie ( tail xs ) )
main = print( fie ([-1,1,-2,2,-3,3,-4,4 ,-5,5] ))
```

그림 2 패턴매칭 에러가 존재하는 프로그램

```
[0, 1, 0, 2, 0, 3, 0, 4, 0, 5,
Program error: {head []}
```

그림 3 HUGS에서의 프로그램 실행결과

```
[0, 1, 0, 2, 0, 3, 0, 4, 0, 5,
Fail: PreludeList.head: empty
```

그림 4 GHC에서의 프로그램 실행결과

그림 2와 같은 간단한 샘플 프로그램을 통해서 레덱스 트레일 디버깅의 유용성과 디버깅이 이루어 지는 과정을 살펴볼 수 있다. 레덱스 트레일 디버깅은 패턴 매칭 오류와 같은 실행시간 오류가 발생했을때 더욱 유용하다. 그 이유는 오류의 원인이 되는 프로그램 상의 위치가 실행시간 오류가 발생한 곳과 밀접한 관계가 있기 때문이다.

그림 2의 프로그램을 현재 가장 널리 쓰이고 있는 Haskell 시스템인 HUGS와 GHC에서 수행했을 때 출력되는 메세지는 각각 그림 3, 그림 4와 같다. 즉, 기존의 Haskell 시스템에서 출력해 주는 오류 메세지는 단지 head함수에서 오류가 발생했다는 정보 뿐임을 알 수 있다. 만약 프로그램의 사이즈가 크다면 오류를 일으킨 head함수가 전체 프로그램에서 어떤 것인지 알 수 있는 정보가 없다.

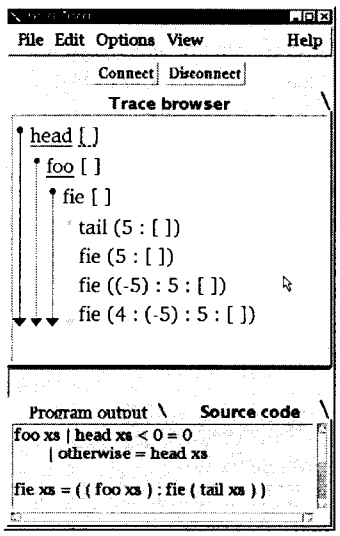

그림 5 레덱스 트레일 탐색기

그림 5는 위의 샘플 프로그램을 레덱스 트레일 디버깅 시스템의 입력으로 넣었을 때 생성되는 트레일 정보를 바탕으로 하여 실제 자바로 구현되어 있는 브라우저를 통해서 디버깅하는 과정을 보여준다.

먼저 프로그램의 실행 오류인 head[]을 시작으로 하여 그 값이 어디에서 부터 나오게 되었는지 확인하기 위해서 브라우저에 나타난 head[]을 마우스로 클릭한다. 그러면 전체 트레일 정보 중에서 현재 head[]의 부모 레덱스인 foo[]를 찾게 되고, 그 결과 head[]은 바로 foo[]의 호출로부터 얻게된 것임을 확인할 수 있다. 여기서 다시 foo[]을 선택하여 마우스 클릭하면 다시 한번 부모 레덱스 검색을 통해서fie[]을 얻게 되는데, 여기서 fie함수의 인자로 얻게 된 []은tail(5:[])로 부터 왔으며 이것은 다시 fie(5:[])로부터 온 것임을 차례로 확인 할 수 있다. 이러한 트레일을 검토해 보면fie함수에 적용되는 리스트의 원소(element)가 하나씩 줄어들어[]이 되었을 경우 에러가 나오는 것을 발견할 수 있고, 따라서 fie함수에 []에 대한 처리를 추가함으로써 오류를 해결할 수 있다.

2.5 레덱스 트레일 디버깅 기법의 평가

알고리즈믹 디버깅 기법과 레덱스 트레일 디버깅 기법의 가장 큰 차이점은, 알고리즈믹 디버깅이 메인 함수에서 시작되는 하향식(top-down)인 반면에 레덱스 트

레일 디버깅 기법은 오류나 잘못된 결과에서부터 디버깅을 시작하는 상향식(bottom-up)이라는 점이다. 이것은 프로그램이 비정상적으로 종료한 경우 문제가 된 그 시점에서 출발하여 오류의 원인을 추적하는 것이 합리적이라는 생각에서 출발한 것이다. 하지만 디버거 사용자가 오류의 위치를 알아내기 위해서는 트레일을 따라서 오류에 접근해 가는 탐구적인 작업(detective work)과 프로그램의 동작에 대한 기본적인 이해가 필수적으로 요구된다. 따라서 프로그램의 동작에 대한 이해를 바탕으로 한다면 알고리즈믹 디버깅보다 쉽게 디버깅 할 수 있지만, 그렇지 않은 경우에는 주어진 결과로부터 오류의 원인이 되는 곳까지 추적하는 작업이 쉽지 않다.

레덱스 트레일 디버깅은 프로그램으로부터 얻게 되는 레덱스 트레일의 사이즈가 크다는 문제점이 있다. 이에 대한 해결책으로 디버거 사용자가 지정하는 크기의 트레일을 부분적으로 생성하는 방법이 제안되었다. 트레일의 크기를 지정한다는 것은 전체 트레일중의 일부분만 생성한 후 먼저 디버깅을 시도하고, 계속해서 나머지 트레일의 생성과 디버깅을 반복한다는 의미이다. 따라서 이것은 트레일의 사이즈를 줄이는 근본적인 해결책이 되지 못한다. 또 다른 방법으로는 Haskell 시스템의 기본 라이브러리인 Prelude와 같이 정확한 동작이 보장되는 특정한 모듈에 한하여 레덱스 트레일이 생성되지 않는 방법이 있다. 이 방법을 적용할 경우에 프로그램의 특성에 따라서 변화의 폭이 있지만 25%~75%정도로 축소된 형태의 레덱스 트레일을 얻은 사례가 있다.

3. 선택적 레덱스 트레일 디버깅

이 장에서는 본 논문에서 제안하는 선택적 레덱스 트레일 디버깅 기법에 대하여 설명한다. 기존의 알고리즈믹 디버깅 기법과 레덱스 트레일 디버깅 기법의 문제점으로 지적되었던 디버깅 정보의 크기를 축소하기 위해서 원하는 부분에만 선택적으로 포커스(focus) 설정을 가능하게 하였고, 서로 분리될 수 있는 디버깅 정보들의 연결성을 유지하여 알고리즈믹 디버깅 기법의 형태인 하향식(top-down)방법과 레덱스 트레일 디버깅 기법의 형태인 상향식(bottom-up)방법을 모두 가능하게 한 선택적 레덱스 트레일 기법에 대해서 알아본다.

3.1 포커스 설정에 의한 레덱스 트레일의 생성

선택적 레덱스 트레일 디버깅 기법의 특징은 디버거 사용자가 디버깅 정보를 원하는 표현식(expression)에 포커스를 설정할 수 있다는 것이다. 어떠한 표현식에 포커스가 설정되면 그 표현식을 계산하기 시작하는 시점에서부터 계산이 끝나는 시점까지 모든 표현식에 대한 트레일을 생성한다. 표현식의 트레일 정보는 추상기계로부터 얻게되는 계산되기 전의 표현식과 계산된 후의 표현식으로 이루어진다.

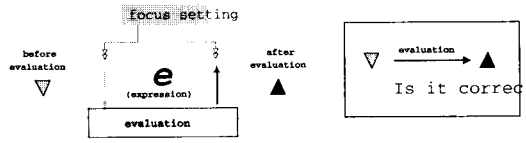

그림 6 포커스가 설정된 표현식의 계산

그림 6에서는 계산되기 전의 표현식을 ▽(before evaluation)으로 표기하고 계산된 후의 표현식을 △(after evaluation)으로 표기하였다. 계산되기 전의 표현식과 계산된 후의 표현식을 비교하면 포커스를 설정한 표현식이 올바른 형태로 계산되었는지를 확인할 수 있게 된다. 메인 함수에서부터 시작하여 모든 표현식에 대하여 이 정보를 트리 구조로 구성한다면, 알고리즈믹 디버깅 기법에서의 EDT를 만들 수 있고 이것을 바탕으로 하향식(top-down)디버깅을 할 수 있다.

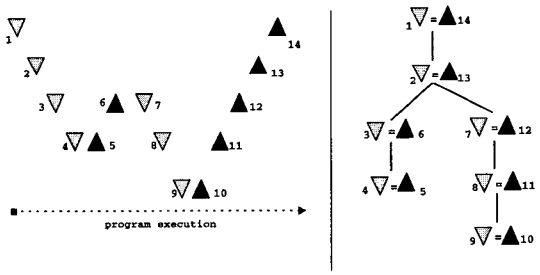

그림 7 프로그램 실행에 따른 전체 트레일

그림 7은 전체 프로그램이 실행하는 동안 표현식이 계산되는 과정을 보여준다. 메인 함수의 값을 계산하기 위해서는 메인 함수에서 호출하는 서브 함수들을 모두 계산해야하고, 서브 함수들도 역시 그 함수에서 호출하는 다른 서브 함수의 결과값이 필요하다. 모든 표현식들이 계산되는 순서대로 연결한다면 하나의 트레일(trail)을 구성할 수 있다. 이것은 레덱스 트레일 디버깅 기법에서의 레덱스 트레일과 동일하고 이 트레일 정보를 기반으로 하여 상향식(bottom-up)디버깅을 할 수 있다.

3.2 선택적 레덱스 트레일의 생성

본 논문에서 제안하는 선택적 레덱스 트레일의 생성은 다음과 같은 의미가 있다. 프로그래머는 일반적으로 프로그램이 오류가 생겼을 때 프로그램에서 오류의 위

치에 대한 직감을 가지고 있다. 따라서 프로그램 전체에 대한 디버깅 정보보다는 부분적으로 선택된 부분에 한하여 디버깅 정보를 저장하게 된다면 전체적인 디버깅 정보의 양을 줄일 수 있게 되어 보다 적은 디버깅 스텝을 이용하여 간단하게 디버깅 할 수 있다. 만일 이 과정에서 오류를 찾지 못했다면 다른 부분에 동적으로 포커스를 다시 설정하거나, 포커스 설정 범위를 넓힘으로써 다시 디버깅 할 수 있다. 이때 프로그램을 다시 컴파일 하는 과정이 없이 재실행에 의해서 변화된 트레일 정보를 얻어낼 수 있다는 장점이 있다.

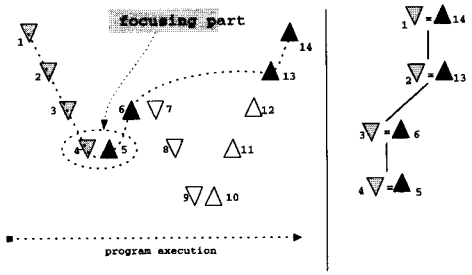

그림 8 포커스에 의해서 생성된 부분적인 트레일

여기서 주의해야 할 점은 그림 8과 같이 포커스를 설정하지 않은 부분에 대해서 트레일을 전혀 생성하지 않는 것이 아니라 단 하나의 트레일만 유지하여 전체적인 프로그램의 흐름이 끊어지지 않게 유지하는 것이다. 단 하나의 트레일만 유지한다는 것은 포커스 영역에 포함되지 않은 부분에 대한 정보는 상대적으로 디버거 사용자의 관심이 적다고 가정하여 그림 6에서의 ▽과 △을 한쌍으로하여 트레일 생성에서 제외시키는 것을 의미한다.

```
start from trail-head;
repeat {
    // common trails
    if ( trail-node->focusing-flag is not TRUE )
    {
        if ( trail-node->evaluation-flag is BEFORE )
            trail-stack.push( trail-node );
        if ( trail-node->evaluation-flag is AFTER )
            trail-statck.pop();
    }
    // focussing trails
    if ( trail-node -> focusing-flag is TRUE )
        trail-stack.push( trail-node );
    trail-node = trail-node->next-node;
} until ( end of trail-node )
```

그림 9 선택적 레덱스 트레일 생성 알고리즘

이 방법을 알고리즘으로 나타내면 그림 9와 같다. 결과적으로 포커스 영역에 진입하기 전에는 단지 ▽만 남게되고 포커스 영역에서 나오게 되면 △만 남게된다. 물론 포커스가 설정된 표현식에 대해서는 그 표현식이 계산되는 과정을 모두 트레일에 포함시킴으로써 세부적인 검토가 가능하도록 한다. 만약 포커스로 설정한 부분들 간의 연결 정보가 없으면 각각의 포커스 영역간의 연관성을 디버거 사용자가 알 수 없게 된다. 이를 보완하여 각각의 포커스로부터 얻어진 레덱스 트레일을 연결시킴으로써 전체적인 트레일의 사이즈에 대한 부담을 줄이고, 또한 알고리즈믹 디버깅을 위한 트리 형태로 계산(evaluation)과정을 표현할 수 있고 프로그램의 시작과 끝이 하나의 트레일로 연결되기 때문에 레덱스 트레일 디버깅이 가능한 형태로 트레일을 유지할 수 있다.

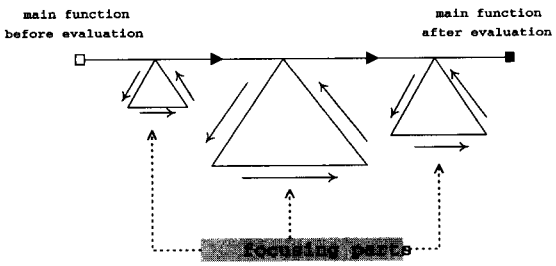

그림 10 전체 프로그램의 실행과 선택된 영역에 대한 부분 트레일

선택적 레덱스 트레일 기법에 의해서 얻게 되는 전체적인 프로그램의 트레일 정보를 그림 10과 같이 표현할 수 있다. 포커스가 설정된 표현식을 만나기 전까지는 ▽ 정보만 유지하고, 포커스가 설정된 표현식을 만나면 ▽과 △ 정보를 함께 유지하고, 포커스가 설정된 표현식을 벗어나면, △ 정보만을 유지한다.

3.3 선택적 레덱스 트레일 디버깅 기법의 평가

프로그래머가 프로그램에 오류가 있음을 발견하고 디버거를 사용할 시점에는 일반적으로 에러의 위치에 대한 예측을 가지고 있다. 선택적 레덱스 트레일 디버깅 기법은 그러한 예측 지점을 포커스로 설정하여 디버깅에 충분히 반영할 수 있다는 장점이 있다. 그리고 포커스를 설정한 부분에 대해서는 상세한 디버깅 정보를 생

성하고 나머지 부분에 대해서는 기본적인 연결을 유지하기 때문에 트레일 추적 단계를 비약적으로 줄일 수 있다.

또한 각 포커스에서 얻은 레덱스 트레일들을 각각 분리시키지 않고 연결하여 알고리즈믹 디버깅 기법의 형태인 하향식(top-down)방법과 레덱스 트레일 디버깅 기법의 형태인 상향식(bottom-up)방법을 모두 가능하게 하였다. 만약 프로그램이 프로그래머의 예상과 다른 결과를 내면서 정상적으로 수행이 완료된 경우에는 알고리즈믹 디버깅 형태의 하향식 탐색을 이용하는 것이 유리하고, 프로그램이 비정상적으로 종료된 경우에는 그 시점에서 오류의 위치를 찾아가는 레덱스 트레일 디버깅 형태의 상향식 탐색이 유리하다.

선택적 레덱스 트레일 기법의 단점은 포커스를 설정하는데 있어서는 레덱스 트레일 디버깅과 마찬가지로 프로그램에 대한 기본적인 이해를 바탕으로 할 경우에 쉽게 오류의 위치를 찾아낼 수 있다는 것이다.

선택적으로 레덱스 트레일을 생성할 수 있는 반면에 만약 포커스를 설정한 영역 안에서 오류의 원인을 찾을 수 없다면 새로운 위치에 포커스를 다시 설정하거나 포커스 범위를 확장시킨 후에 다시 디버깅을 시작해야 한다. 하지만 프로그램을 다시 컴파일할 필요없이 실행시간(run-time)에 동적으로 포커스의 설정 및 해지가 가능하기 때문에 포커스 재설정에 따르는 컴파일 타임 오버헤드를 줄일 수 있다.

4. 구현 및 실험

이 장에서는 레덱스 트레일 기법에 의해서 구현된 디버깅 시스템에 대하여 알아본다. 먼저 트레일 정보를 생성하는 추상기계와 그 정보를 가지고 실제로 디버깅을 수행하는 탐색기를 각각 설명하고, 선택적 레덱스 트레일 디버깅 기법의 성능을 몇가지 실험을 통해서 알아본다.

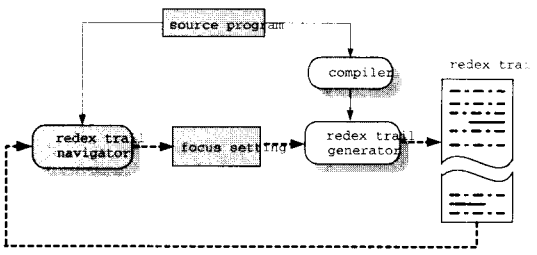

그림 11 선택적 레덱스 트레일 디버깅 시스템의 구조

4.1 레덱스 트레일 생성기

본 논문에서는 레덱스 트레일을 생성기로써, Josephs가 제시한 지연계산 기반 언어의 의미식에 Wand-Clinger 방법을 적용하여 얻은 추상기계를 사용하였다[14]. Wand와 Clinger는 언어의 명세로 denotational semantics 기반 언어 의미로부터 컴파일러를 구성해 내는 방법론을 제안했다. 이 방법론은 우선 언어의 의미를 분석하여 컴파일시간에 결정되는 정적인 요소와 실행시간에 결정되는 동적인 요소로 구분한다. 그리고 정적인 요소를 이용해서 주어진 입력 언어(source language)를 출력 언어(target language)로 변환하는 컴파일러를 구성하고, 동적인 요소를 이용해 출력 언어를 수행하는 추상기계를 구성한다.

언어 의미식에서 컴파일 시간에 결정할 수 있는 정적인 요소로 심볼 테이블(symbol table)과 기본 블럭(basic block)이 있다. 심볼 테이블은 식에 나타난 변수 이름을 오프셋 주소(offset address)로 변환하기 위한 것이고, 기본 블럭은 동일한 실행시간 환경 하에서 주어진 식을 계산하고 그 결과를 스택에 저장하는 명령어들을 나열한 것이다. 컴파일러는 정적인 요소인 심볼 테이블을 계산한 후 실행할 기본 블럭을 받아서 또 다른 기본 블럭을 내는 형태로 구성되어 있다. 어떤 표현식을 컴파일하여 결과로 얻은 기본 블럭은 실행시간에 디스플레이, 스택, 실행시간 컨티뉴에이션(실행시간 컨티뉴에이션은 함수 호출시 필요한 리턴 주소를 저장하는 스택 역할을 한다.), 메모리를 받아 결과를 내고, 추상기계는 언어 의미식에서 동적인 요소를 받아 명령어를 수행하는 형태이다. 본 논문에서는 이용한 컴파일러와 추상기계는 Haskell 언어로 제작되었고, 웹을 통해서 공개되어 있다[15].

4.2 레덱스 트레일 탐색기

이 논문에서 구현된 선택적 레덱스 트레일 탐색기의 사용자 화면은 그림 12와 같다. 전체 윈도우는 프로그램 소스 브라우저와 레덱스 트레일의 브라우저, 알고리즈믹 디버깅 브라우저로 구분된다.

소스 브라우저는 포커스를 설정할 수 있는 각각의 표현식을 적색 박스 형태로 표시한다. 디버거 사용자는 포커스 설정을 원하는 표현식을 마우스 클릭으로 선택할 수 있으며 이미 선택되어 있는 표현식을 다시 선택할 경우에는 포커스가 해지된다.

레덱스 트레일 브라우저에서는 포커스에 의해서 선택된 레덱스 트레일을 한단계씩 추적할 때에 현재의 레덱스 트레일이 어떤 것인지 알려주는 역할을 한다. 소스 브라우저에서는 현재 추적중인 레덱스 트레일을 청색

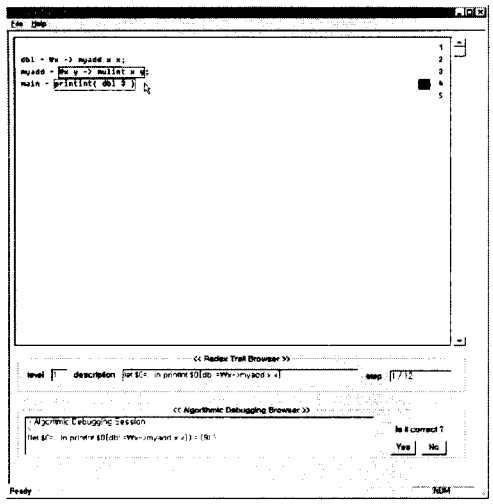

그림 12 선택적 레덱스 트레일 탐색기

박스 형태로 표시해서 현재 추적중인 레덱스 트레일과 실제 프로그램 소스를 연관시킨다.

알고리즈믹 디버깅 브라우저에서는 선택적 레덱스 트레일에 의해서 생성된 EDT를 기반으로 디버거 시스템이 질문을 한다. 디버거 사용자는 그에 대한 '예', '아니오' 응답을 하고, 디버거 시스템은 디버거 사용자의 응답을 기반으로 오류가 있는 표현식을 알려준다.

4.3 선택적 레덱스 트레일 디버깅이 수행되는 과정

레덱스 트레일 생성기와 레덱스 트레일 탐색기로 구성된 전체 디버깅 시스템의 구조는 그림 11과 같다. 선택적 레덱스 트레일 디버깅 시스템을 이용하여 실제로 디버깅을 수행하는 과정을 샘플 프로그램을 중심으로 설명한다.

```
dbl = \x -> myadd x x;
myadd = \x y -> mulint x y;
main = printint( dbl 3 )
```

위와 같은 프로그램을 추상기계의 입력으로 넣어서 모든 트레일을 생성할 경우 아래와 같은 레덱스 트레일을 얻을 수 있다. 트레일의 인덱스는 편의상 추가하였고, 레덱스 트레일 정보는 플래그(flag)와 레덱스 정보의 두가지로 구성된다. 플래그는 'v'와 '^'의 두가지 형태가 있는데,'v' 표시는 표현식을 계산하기 전의 형태를 의미하고,'^' 표시는 표현식이 계산된 후의 형태를 의미한다. 각각의 트레일 정보는 레덱스의 값이 평가되는 과정을 순차적으로 표현하고 있다. 즉, 트레일 번호 1의 내용은 결과값이 저장되는 메모리 주소 0 번지 값을 출력하기 위한 트레일이고, 이것과 쌍을 이루는 트레일 번호 28의 9라는 결과가 출력된다. 트레일 번호 2의 내용은 메모리 주소 0번지에 저장되는 값이 (Int(#3))2 값에 의해서 정해지는 것을 의미하는데 (#3)2 은 정수 3을 나타내는 추상기계의 내부적인 표현이다. 트레일 번호 3은 dbl함수에 정수 3을 적용시키는 것을 의미하는데 이곳에서 dbl함수의 정의는 [dbl:=\x->myadd x x]라는 것을 알 수 있다. 이와 같은 방식으로 각각의 레덱스 트레일 정보는 전체 프로그램의 결과값을 얻기 위한 일련의 과정을 순차적으로 표현하고 있다.

```
 1 v : let $0=.. in printInt $0[dbl:=\x->myadd x x]
 2 v : $0[$0:=dbl(Int (#3))]
 3 v : dbl(Int (#3))[dbl:=\x->myadd x x]
 4 v : dbl[dbl:=\x->myadd x x]
 5 v : \x->myadd x x[myadd:=\x y->*Int x y]
 6 ^ : \x->myadd x x
 7 ^ : \x->myadd x x
 8 v : myadd x x[myadd:=\x y->*Int x y,x:=Int (#3)]
 9 v : myadd[myadd:=\x y->*Int x y]
10 v : \x y->*Int x y
11 ^ : \x y->*Int x y
12 ^ : \x y->*Int x y
13 v : *Int x y[x:=x,y:=x]
14 v : x[x:=x]
15 v : x[x:=Int (#3)]
16 v : Int (#3)
17 ^ : Int #3
18 ^ : Int #3
19 ^ : Int #3
20 v : y[y:=x]
21 v : x[x:=Int #3]
22 ^ : Int #3
23 ^ : Int #3
24 ^ : Int #9
25 ^ : Int #9
26 ^ : Int #9
27 ^ : Int #9
28 ^ : 9
```

그러나 생성되는 모든 레덱스 트레일 정보의 양이 많기 때문에 디버거 사용자는 포커스 설정을 고려할 수 있다. 샘플 프로그램의 myadd함수의 동작을 살펴보기 위해서 레덱스 트레일 브라우저에서 myadd함수를 포함한 박스를 선택하고 레덱스 트레일을 생성하면 아래와 같이 축소된 레덱스 트레일 정보를 얻게 된다.

```
 1 v : let $0=.. in printInt $0[dbl:=\x->myadd x x]
 2 v : $0[$0:=dbl(Int (#3))]
 3 v : dbl(Int (#3))[dbl:=\x->myadd x x]
 8 v : myadd x x[myadd:=\x y->*Int x y,x:=Int (#3)]
```

```
 9 v : myadd[myadd:=\x y->*Int x y]
10 v : \x y->*Int x y
11 ^ : \x y->*Int x y
12 ^ : \x y->*Int x y
25 ^ : Int #9
26 ^ : Int #9
27 ^ : Int #9
28 ^ : 9
```

이와 같이 선택적으로 생성된 트레일을 순차적으로 살펴보면 프로그램 오류의 위치를 좀더 쉽게 찾아낼 수 있다. 트레일 번호 8과 9의 내용을 보면 myadd 함수가 어떻게 치환되는지 알 수 있는데 myadd 함수의 정의는 프로그래머의 의도인 [myadd:=\x y->+Int x y]가 아니라, [myadd:=\x y->*Int x y]로 되어 있기 때문에 dbl(3)의 값이 6으로 계산되지 않고, 9로 계산된 것이다.

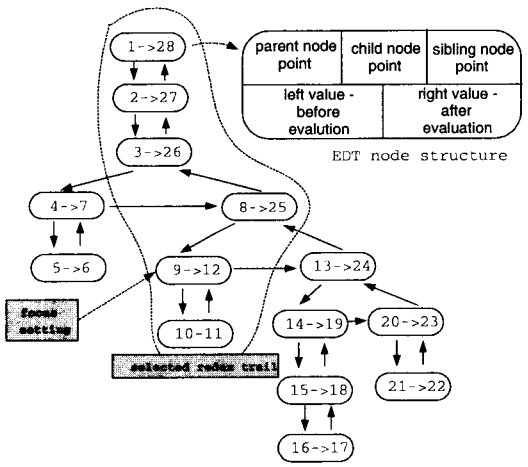

그림 13 EDT노드의 구조와 선택적 레덱스 트레일 정보로부터 생성된 EDT

주어진 샘플 프로그램은 프로그램이 정상적으로 수행이 되지만 예상하지 않은 결과를 출력하는 프로그램이다. 따라서 상향식의 레덱스 트레일 디버깅 보다는 하향식의 알고리즈믹 디버깅을 이용하는 것이 더 적합하다. 생성된 레덱스 트레일을 바탕으로 알고리즈믹 디버깅을 가능하게 하기 위해서는 EDT를 생성해야 한다. 레덱스 트레일 정보로부터 EDT를 생성하는 세부적인 알고리즘은 그림 14에 포함하였다, 생성된 EDT와 EDT를 구성하는 노드의 구조는 그림 13에서 볼 수 있다. EDT에서 각각의 노드에 실제로 포함되는 레덱스 정보는 레덱스 트레일 정보에서의 인덱스로 대신했다. 샘플 프로그램의 오류를 알고리즈믹 디버깅에 의해서 찾아내는 일련의 과정은 다음과 같다.

```
> Algorithmic Debugging Session
>
> (let $0=.. in printInt $0[dbl:=\x->myadd x x]) = (9) ?
> No
> ($0[$0:=dbl(Int (#3))]) = (Int #9) ?
> No
> (dbl(Int (#3))[dbl:=\x->myadd x x]) = (Int #9) ?
> No
> (myadd x x[myadd:=\x y->*Int x y,x:=Int (#3)]) = (Int
#9) ?
> No
> (myadd[myadd:=\x y->*Int x y]) = (\x y->*Int x y) ?
> Yes
> Error Located in this node.
>
```

```
start from EDTroot;
repeat {
{
  if ( redex_trail_flag is 'v' )
  {
    make child-node;
    // left value setting
    node->Lvalue = redex_trail_info;
    // Parent Setting
    node->parent = cur_node;
    // First Child Create
    if ( cur_node->child is empty )
      cur_node->child = node;
    else { // Next Child Create
        // Last Child Find
        while( sibling node is exist )
            move to sibling node;
        // Attach Current Node to Sibling Node
        last-sibling-node->sibling = node;
    }
    cur_node = node;
  }
  else { // redex_trail_flag is '^'
    // Right Value Setting
    cur_node->Rvalue = redex_trail_info;
    // Move To Parent Node
    cur_node = cur_node->parent;
  }
} until ( the number of trail-node )
```

그림 14 레덱스 트레일로부터 EDT를 생성하는 알고리즘

4.4 실험 및 평가

본 논문에서 실험 결과의 기준은 두가지로 정하였다. 첫째, 선택적 레덱스 기법에 의한 레덱스 트레일의 축소이고, 둘째, 축소된 레덱스 트레일에 의한 프로그램의 동작 속도 향상이다. 본 논문 제안하는 아이디어의 핵심은 기존의 디버깅 방법들이 생성하는 레덱스 트레일 정보를 저장하는 오버헤드가 크고, 또한 그에 따라서 레덱스 트레일을 생성하는데 걸리는 시간에 대한 오버헤드가 크기 때문에 이 두가지 문제점에 대한 해결책을 찾는데 있다. 물론 선택적 레덱스 트레일 기법을 적용할 경우에는 프로그램 안에서 선택된 포커스 부분이 차지하는 부분의 비율에 따라서 줄어드는 레덱스 트레일 정보를 생성하는 것은 자명한 것이고, 또한 포커스를 선택하는 부분에 따라서 결과는 달라지기 때문에 이에 대한 실험적 평가는 불필요하다고 생각할 수 있다. 그러나 프로그래머가 원하는 부분에 포커스를 설정하여 디버깅을 시작할 경우 실험적으로 어느 정도의 레덱스 트레일의 축소 효과가 있는지에 대한 조사가 유용하다고 생각하였기 때문에 본 논문에서와 같은 실험 결과를 제시하였다.

표 1 선택적 레덱스 트레일 기법에 의한 트레일의 축소

trail / program	full(redex number)	selected(redex number)	ratio(selected / full)
tree sort	1216	192	16%
fibonacci(7th)	1394	492	35%
factorial(50th)	1424	612	43%

표 1은 선택적 레덱스 트레일 디버깅 기법을 샘플 프로그램에 적용한 결과를 보여준다. 이 결과에 의하면 선택적 레덱스 트레일은 전체 트레일에 비교해서 16~43%정도로 축소된 트레일을 생성한다는 것을 확인할 수 있다.

이 실험을 통해서 tree sort 프로그램과 같이 여러 개의 함수나 모듈의 조합으로 이루어진 프로그램에서 선택적 레덱스 트레일 기법의 성능이 우수하지만, fibonacci 수를 구하는 프로그램이나 어떤 수의 factorial 값을 구하는 프로그램처럼 집중적인 재귀호출로 구성된 프로그램에서는 포커스 설정을 통한 선택적 레덱스 트레일 기법이 레덱스 트레일 축소에 큰 효과를 거둘 수 없다는 것을 쉽게 짐작할 수 있다. 이것은 포커스가 설정된 부분의 실행이 전체 프로그램 실행에서 차지하는 비중이 크다면 비록 포커스를 적게 설정하더라도 선택적 레덱스 트레일의 결과로 얻게되는 트레일의 사이즈가 커지게 되기 때문이다. 예를 들어 재귀호출 루틴에 하나의 포커스가 설정하더라도 재귀호출의 시작과 함께 트레일이 생성이 시작되고 재귀호출의 종료와 함께 트레일의 생성이 종료되어 그 결과로 얻게되는 레덱스 트레일의 사이즈는 커질 수 있다.

재귀 호출의 경우에는 호출 인자값만이 바뀌면서 반복적으로 동일한 동작이 이루어지기 때문에 만일 재귀호출의 깊이를 제한하여 레덱스 트레일을 생성할 수 있다면 트레일 크기의 축소에 효과를 거둘 수 있을 것이다.

표 2 선택적 레덱스 트레일 기법에 의한 속도 향상

trail / program	full(sec)	selected(sec)	ratio(selected / full)
tree sort	1.37	0.56	41%
fibonacci(7th)	1.05	0.34	32%
factorial(50th)	1.08	0.36	33%

표 2는 선택적 레덱스 트레일 디버깅 기법을 샘플 프로그램에 적용했을 때 프로그램 실행 속도에 대한 실험 결과이다. 선택적 레덱스 트레일의 생성 기법은 전체 트레일을 생성하면서 실행했을 경우와 비교하여 대략 32%~41%정도로 속도를 단축시키는 결과를 얻었다.

5. 결론 및 향후과제

함수형 프로그래밍 언어는 명령형 언어에 비하여 많은 장점을 가지고 있지만, 상대적으로 디버깅 환경이 매우 빈약하고, 또한 현존하는 함수형 언어 디버깅 기법들이 기존의 명령형 프로그래밍 언어에서 유용한 기법들에 기반을 두고 있기 때문에 함수형 언어에 그대로 적용되어 사용되기에는 무리가 있었다.

함수형 언어에서 이용되는 대표적인 디버깅 기법으로 하향식(top-down)접근 방법인 알고리즈믹 디버깅 기법과 상향식(bottom-up)접근 방법인 레덱스 트레일 디버깅 기법에 대하여 알아보았다. 알고리즈믹 디버깅 기법은 디버깅 시스템이 요구하는 질문에 대한 디버거 사용자의 정확한 답변을 전제로 한다면 오류의 위치를 정확하게 알아낼 수 장점이 있다. 그러나 디버깅에 이용되는 트리의 크기에 대한 부담과 그에 따라 디버깅 시스템으로부터의 질문의 수도 많아진다는 단점이 있다. 레덱스 트레일 디버깅 기법은 프로그램의 결과에서부터 디버깅

을 시작하는 방법이기 때문에 실행시간 오류에 대한 디버깅에 유리하다. 하지만 디버거 사용자는 레덱스 트레일의 추적 단계를 직접 선택해야 하기 때문에 주어진 결과로부터 오류의 원인이 되는 곳까지 찾아가는 것이 쉽지 않고, 디버깅을 위한 레덱스 트레일의 사이즈가 크다는 단점이 있다.

본 논문에서 제안한 선택적 레덱스 트레일 디버깅 기법은 디버거 사용자의 프로그램 오류에 대한 직감을 바탕으로 포커스를 설정할 수 있게 하였다. 그리고 포커스가 설정된 부분에 한하여 세부적인 트레일을 생성하여 트레일 사이즈에 대한 부담을 줄였다. 선택적 레덱스 트레일 디버깅 기법은 단지 레덱스 트레일의 축소에만 의미가 있는 것은 아니다. 표현식이 계산되기 전과 계산된 후의 트레일 정보를 하나로 묶어서 EDT를 구성하였고, 이것을 기반으로 알고리즈믹 디버깅 기법에서의 하향식 디버깅을 가능하게 하였다. 선택적 레덱스 트레일 디버깅 시스템은 레덱스 트레일을 생성하는 추상기계와 생성된 트레일 정보로부터 디버깅을 수행하는 탐색기로 분리하여 구현하였다. 추상기계에서 동작하는 디버깅 기법은 프로그램 변환 방식을 이용한 디버깅 기법과 비교하여 상대적으로 이식성이 부족하다. 그러나 포커스 설정을 바꿀 때마다 프로그램을 다시 컴파일할 필요가 없고 재실행을 통해서 새로운 트레일 정보를 얻을 수 있다는 장점이 있다. 구현된 디버깅 시스템을 이용해서 실험한 결과 프로그램의 특성과 포커스 설정에 따라서 차이가 있지만 전체 레덱스 트레일에 비교하여 16%~43%정도로 축소된 트레일을 얻었고, 속도 측면에서는 32%~41%정도로 향상된 결과를 얻었다.

본 논문에서 제시한 디버깅 시스템에 대해서 앞으로 진행되어야 할 연구 과제는 다음과 같다. 우선, 본 논문에서 제시한 방법의 성능 향상을 위한 보완이 필요하다. 즉, 선택적 레덱스 트레일 디버깅 기법과 함께 정확한 동작이 보장되는 모듈이나 함수의 경우에 트레일을 만들지 않는 방법을 적용시키는 것이다. 이 방법은 속도 측면에서의 부담을 고려해야 하지만 좀더 축소된 디버깅 정보를 얻을 수 있을 것이다. 또한, 재귀호출 함수에 포커스를 설정할 때 발생하는 문제점에 대한 고려가 필요하다. 간단한 해결책으로, 재귀호출 함수의 호출 깊이를 제한하여 부분적인 트레일을 생성하는 기법이 이용될 수 있을 것이다. 마지막으로, 디버거 정의 언어를 설계하여 프로그램을 이루고 있는 각각의 표현식으로부터 어떤 정보를 추출할 것인지를 세부적으로 기술하게 할 수 있다. 예를 들면, 대상 프로그램으로부터 여러가지 다른 파라미터(속도, 메모리 요구량)의 정보가 포함된 트레일 정보를 얻을 수도 있다. 이것은 추상기계를 디버거 사용자의 요구사항에 따라서 수정할 필요없이 디버거 정의 언어를 바탕으로 다양한 형태의 디버깅 정보를 얻을 수 있는 범용적인 디버거의 구성을 가능하게 할 것이다.

참 고 문 헌

[1] John Hughes.Why Functional Programming Matters. *Computer Journal, 32(2), 1989.*

[2] Jonathan E.Hazan and Richard G.Morgan. The Location of Errors in Functional Program. In Peter Fritzson, editor, *Automated and Algorithmic Debugging*, volume 749 of*Lecture Notes in Computer Science, Linkoping, Sweden, May 1993.*

[3] Henrik Nilsson. A Declarative Approach to Debugging for Lazy Functional Languages. Linkoping Studies in Science and Technology Thesis No 450, 1994.

[4] Jan Sparud and Colin Runciman. Tracing Lazy Functional Computations Using Redex Trails. In *Proc. 9th International Symposium on Programming Languages, Implementations, Logics and Programs, 1997.*

[5] Cordelia V.Hall and John T.O'Donnell. Debugging in Applicative Languages. In *Lisp and Symbolic Computation, 1988.*

[6] I.Toyn and C.Runciman. Adapting combinator and SECD machines to display snapshots of functional computations. *New Generation Computing, 4(4):339-363, 1986.*

[7] Jan Sparud. Transformational Approach to Debugging Lazy Functional Programs. Licentiate Thesis, Department of Computer Science, Chalmers University of Technology, S-412 96, Goteborg, Sweden, February 1996.

[8] E.Y. Shapiro. *Algorithmic Program Debugging, MIT Press, May 1982.*

[9] Jan Sparud and Colin Runciman. Partial Redex Trails of Large Functional Computations. *Implementation of Functional Languages, 8th International Workshop, 1997.*

[10] Henrik Nilsson and Peter Fritzson. Algorithmic Debugging for lazy functional languages. *Journal of Functional Programming, 4(3),1994.*

[11] Rickard Westman and Peter Fritzson. Graphical User Interfaces for Algorithmic Debugging. In Peter Fritzson, editor, *Automated and Algorithmic Debugging*, volume 749 of*Lecture Notes in Computer Science, Linkoping, Sweden, May 1993.*

[12] Henrik Nilsson and Peter Fritzson. Lazy Algorithmic Debugging: Ideas for Practical Imple-

mentation. Automated and Algorithmic Debugging, volume 749 of Lecture Notes in Computer Science, pages 117-134, Linkoping, Sweden, May 1993.

[13] Henrik Nilsson, Jan Sparud. The evaluation dependence tree as a basis for lazy functional debugging. *Journal of Automated Software Engineering, 4(2):152-205, April 1997.*

[14] 최광훈, 한태숙. 지연계산 기반 추상기계 유도 : 지연계산 기반 언어의 의미에 Wand-Clinger 방법의 적용. '98 추계 학술발표논문집, 제10회 정보과학회 충청지부 학술발표회, 1998년 11월.

[15] 지연계산 기반 언어의 가상기계 http://pllab.kaist.ac.kr/~khchoi/haskell/absmachine/

[16] Anthony J.Field, Peter G.Harrison. *Functional Programming, Addison-Wesley 1988.*

[17] The Glasgow Haskell compiler. http://www.dcs.gla.ac.uk/fp/software/ghc/.

[18] Lee Naish and Tim Barbour. Towards a portable lazy functional declarative debugger. Tehcnical Report 95/27, Department of Computer Science, Univercity of Melbourne, Australia, 1995.

[19] Jan Sparud. Towards a Haskell debugger. In *Functional Programming Languages and Computer Architecture, 1995.*

[20] Andrew Tolmach and Andrew W.Appel. A Debugger for Standard ML. *Journal of Functional Language, 1(1):1-000, January 1993.*

부록 : 예제 프로그램

*밑줄로 표기한 부분에 포커스를 설정하였다.

1. Tree sort

```
data Tree a = Tip a | Tree a :^: Tree a deriving Show
sort [] = []
sort (x:xs) = insert x (sort xs)
insert x [] = [x]
insert x xxs@(x':xs) | x <x' = x : xxs
    otherwise = x' : insert x xs
sortTree t = t_tree
    where
        (t_stips, t_ssorted, t_tree) = sortTree' t t_itips t_isorted
        t_itips = []
        t_isorted = sort t_stips
sortTree' (Tip a) t_itips t_isorted =
        (t_stips, t_ssorted, t_tree)
    where
        t_stips = a : t_itips
        t_ssorted = tail t_isorted
        t_tree = Tip (head t_isorted)
sortTree' (l:^:r) t_itips t_isorted =
        (t_stips, t_ssorted, t_tree)
    where
        (l_stips, l_ssorted, l_tree) = sortTree' l l_itips l_isorted
        (r_stips, r_ssorted, r_tree) = sortTree' r r_itips r_isorted
        r_itips = t_itips
        l_itips = r_stips
        t_stips = l_stips
        l_isorted = t_isorted
        r_isorted = l_ssorted
        t_ssorted = r_ssorted
        t_tree = l_tree :^: r_tree
aTree = (Tip (7::Int) :^: (Tip 2 :^: Tip 5)) :^: (Tip 3 :^: Tip 1)
main = print (sortTree aTree)
```

2. Factorial function

```
fac 0 = 1
fac n = n * fac (n-1)
main = print (fac 50)
```

3. Fibonacci function

```
nfib 0 = 1
nfib 1 = 1
nfib n = nfib (n-1) + nfib (n-2)
main = print( nfib 7 )
```

박 회 완
1997년 2월 동국대학교 컴퓨터공학과 졸업. 1999년 2월 한국과학기술원 전산학과 졸업(공학석사). 1999년 3월 ~ 현재 한국과학기술원 전자전산학과 전산학전공 박사과정 재학중. 관심분야는 함수형 언어, 디버거, 프로그램 슬라이싱, 자바임

한 태 숙
1976년 서울대학교 전자공학과 졸업. 178년 한국과학기술원 전산학과 졸업. 1990년 Univ. of North Carolina at Chapel Hill 졸업. 현재 한국과학기술원 전자전산학과 전산학전공 부교수. 관심분야는 프로그래밍 언어론, 함수형 언어임