ROPAS
Research On Program Analysis System
National Creative Research Initiative Center 1998-2003
Programming Research Laboratory, School of Computer Science & Engineering
Seoul National University

ROPAS MEMO
2006-30
July 18, 2006

프로그램 분석에서 넓히기(widening)로 인한 정확도 저하를 보완하는 할일하기(worklist) 알고리즘

이상우

서울대학교 대학원
전기컴퓨터 공학부 컴퓨터공학 전공
석사학위 논문

지도교수 이광근

요 약

본 논문은 프로그램 분석중 넓히기(widening)가 할일하기(worklist) 알고리즘과 연계된 경우 분석에 대한 정확성을 저하시키는데 착안하여 그 해결책을 제시한다.

넓히기(widening)는 무한히 반복되는 프로그램을 유한한 시간내에 분석하기 위해 사용된다. 무한 반복문을 프로그램 분석에서 분석할 경우, 그 프로그램의 의미대로 분석을 수행하다보면 프로그램이 계속 반복되어 분석을 종료하지 않을 경우가 생긴다. 이 경우 넓히기를 이용하여 무한히 수행되는 반복문을 빠져 나올 수 있다. 무한히 수행될 가능성이 있는 분석에 넓히기를 통해 무한이 반복되는 분석을 막는다.

할일하기(worklist) 알고리즘을 사용한 분석에 넓히기를 적용시킨다면 정확성을 떨어뜨리는 분석이 실행될 수 있다. 프로그램 분석중 프로그램 방정식을 세워 그 방정식을 풀어갈 때, 식이 합쳐지는(join) 부분에서 할일하기 알고리즘이 넓히기 연산과 함께 작용하면서 고정점에 도달하는 식임에도 불구하고 넓히기를 적용함으로써 그 부분의 방정식 값에 정확도를 저하시키는 결과를 초래한다.

이렇게 정확도가 낮아지는 요인은 합쳐지는 두 식이 필요이상으로 할일하기 알고리즘에 의해 촘촘히 계산되기 때문이다. 큰 폭으로 건너뛰어 방정식을 계산하면 고정점(fixpoint)에 도달할 식을 할일하기(worklist) 알고리즘의 촘촘한 계산이 넓히기 수행을 유발하여 정확도를 떨어뜨리게 되는 것이다.

이에 대한 해결책으로 active-worklist와 wait-worklist 두 개의 worklist로 방정식을 푼다. 기본원리는 방정식의 합쳐지는 부분에서 한쪽 식에서의 값의 변화를 감지하지 않고, 양쪽 식의 값들이 정해질때까지 기다린후 넓히기를 적용하여 분석의 정확도를 향상시키는 수정된 할일하기(worklist) 알고리즘을 제시한다.

1 서론

할일하기(worklist) 알고리즘이 적용된 분석기에서 넓히기(widening)가 동작할 때 어떠한 문제를 발생하는가에 대해 살펴보기 앞서, 배경지식으로 프로그램 분석 방정식, 넓히기, 그리고 할

일하기 알고리즘에 대해서 살펴보고, 이들 사이의 연계가 어떤 문제를 발생시키는지에 대해서 알아본다.

1.1 배경지식[7]

프로그램을 실행전에 미리 검증해보는 기술로 프로그램 분석 기술이 각광받고 있다. 프로그램이 과연 우리가 바라는 대로 실행될 것인가? 우리가 바라는 대로 프로그램이 실행될 것인지 미리 프로그램을 검증할 방법은 있는가? 프로그램 속에 녹아 들어있는 도출된 버그나 프로그램 속에 내재된 버그를 실행전에 미리 찾을 방법이 있는가? 이에 대한 해답으로 프로그램 분석 기술이 점점 발전해오고 있다.

지난 과거부터 프로그램의 버그를 줄이기 위해서 많은 노력이 있어 왔다. 컴퓨터의 발달과 함께 오랜시간 전부터 프로그램의 버그를 줄일 수 있는 기술이 많이 개발되어 왔다. 1970년대에는 프로그램이 대상 프로그램언어에 맞게 형식대로 기술되어 있는지 검증하는 구문검증기술(parsing)이 등장했다. 이 기술로 프로그램머들이 짠 프로그램을 대상 언어의 형식대로 기술하였는지 일일이 검증할 필요가 없게 되었다.

이후 등장한 기술은 바로 타입검증(type checking)이다. 구문검증기술만으로 프로그램의 오류를 걸러낼 수 없었다. 프로그램 언어의 형식대로 올바르게 기술하였을지라도 실제로 프로그램이 의도대로 수행되지 않는 경우가 발생했다. 이런 프로그램들을 안전하게 걸러내기 위해서 1990년대에 타입검증 기술이 등장하게 된다. 타입에 맞지 않는 프로그램들은 실행중에 계산을 할 수 없게 된다. 이러한 프로그램은 예측할 수 없는 방향으로 프로그램이 실행되거나 아무도 찾을 수 없는 오류로 프로그램의 수행을 멈추게 한다. 프로그램의 모든 값들은 그것이 정의된 타입에 맞는 값들을 가져야 한다. 프로그램이 항상 그러한 성질을 만족하면서 실행될 것인지 자동으로 검증해 주는 것이 바로 타입검증이다.

이후, 프로그램의 정교한 오류까지 검증하기 위한 기술로 자동증명기술(theorem proving)과 정적분석 기술(static analysis)이 현재까지 발전되어 오고 있다. 이러한 기술들은 분석하고자 하는 프로그램의 성질을 정교한 논리식으로 정의한다. 대상 프로그램이 그러한 정교한 논리식을 만족하는지 그렇지 않은지를 여러 방법을 통해 찾는다. 그 방법 중에는 요약해석(abstract interpretation), 모델체킹(model checking) 등이 존재하고 이러한 기술들은 현재까지 꾸준히 발전해오고 있다.

정적분석은 프로그램 분석(static program analysis)[1,3,4]이라고도 한다. 프로그램 분석은 프로그램이 실행중에 가지는 성질을 실행전에 자동으로 안전하게 어림잡는 일반적인 방법이다. 안전하게 어림잡는다는 뜻은 프로그램을 직접 수행시키지 않고서 정확하게 프로그램이 어떠한 수행을 한다고 말할 수 없으므로 직접 수행시켜 발생하는 결과를 포함하여 대략 어림잡는다는 뜻이다. 실제결과를 모두 포함하여 어림잡기 때문에 실제결과 이외의 값들이 포함하게 된다. 이런 실제이외의 값들이 정적분석을 이용하여 분석기를 설계하여 수행할때 나오는 거짓 알람(false alaram)을 발생시킨다.

1.1.1 프로그램 분석 방정식

프로그램 분석에서 분석 알고리즘은 다음과 같이 프로그램 C에 대한 방정식을 세우는 것이다.

$$\vec{X}^{k+1} = F_C \vec{X}^k$$

$$F_C : D \rightarrow D$$

$$\begin{array}{rcl} F_C & = & \lambda \vec{X}.\langle f_1 \vec{X}, \cdots, f_n \vec{X} \rangle \\ \vec{X} & = & \langle X_1, \cdots, X_n \rangle \end{array}$$

X_n는 프로그램 C의 부품 C_n의 의미를 뜻하고 F_C는 프로그램의 실행함수이고 k는 itertation 횟수이다. 방정식들은 위와 같이 정해진다. 위 방정식의 해는 방정식의 변수가 고정점에 도달할 때까지 iteration을 증가하여 구할 수 있다. 이 방정식의 해는 다음의 윗뚜껑(LUB, least upper bound)을 계산하면 된다.

$$\bigsqcup_{k \in \mathbb{N}} {F_C}^k(\perp_1, \perp_2, \cdots, \perp_n)$$

정리하면, $\vec{X} = \langle X_1, \cdots, X_n \rangle$ 일때

$$\begin{array}{rcl} \vec{X}^0 & = & < \perp_1, \perp_2, \cdots, \perp_n > \\ \vec{X}^{k+1} & = & F_C \ (\vec{X}^k) \qquad (k > 0) \end{array}$$

다음을 구하면 프로그램의 분석 방정식을 정의할 수 있고, 그 해를 찾을 수 있다.

1.1.2 할일하기 알고리즘(worklist algorithm)

할일하기 알고리즘[2]을 요약하면, worklist에 해야할 일을 저장하고 각 iteration 단계를 지날 때마다 worklist에서 일을 선택하여 수행하고, 선택된 일은 worklist에서 삭제하는 알고리즘이다. 선택된 일을 수행할 때에 그 선택된 일의 값의 변화에 영향을 받는 일들이 새로이 worklist에 저장된다. 이 과정은 더 이상 해야 할 일들이 없을 때까지, 다시말하면 worklist가 비어있을때까지 수행된다.

할일하기 알고리즘은 표 1과 같다.

- empty는 빈 worklist이다.
- insert((x⊒ t),W) 은 W에 제약 $x \sqsupseteq t$ 이 더해진 worklist을 리턴한다.
  즉 W:=insert($(x \sqsupseteq t), W$).
- extract(W)은 첫번째 컴포넌트가 worklist에서 제약 $x \sqsupseteq t$ 이고, 두번째 컴포넌트가 $x \sqsupseteq t$을 제거함으로써 얻어진 worklist인 쌍을 리턴한다.
  즉, (($x \sqsupseteq t$),W):=extract(W).

아이락[5]은 할일하기 알고리즘[2]을 이용하여 방정식을 푼다. 아이락은 프로그램 분석 기술을 이용하여 할당한 메모리 영역을 벗어나서 접근하는 C 프로그램 오류의 위치를 미리 모두 자동으로 찾아주는 분석기이다. 아이락은 방정식의 모든 요소를 매 iteration마다 계산하지 않고 할일하기 알고리즘으로 방정식의 변한 값들을 추적하여 그와 관계된 방정식만을 계산하는 효율적인 방법으로 분석한다.

```
INPUT:    A system S of constraints:  x_1 ⊒ t_1, ··· , x_N ⊒ t_N
OUTPUT:   The least solution:  Analysis
METHOD:   Step 1:  Initialisation (of W, Analysis and infl)
                   W:=empty;
                   for all x ⊒ t in S do
                     W:=insert((x ⊒),W)
                     Analysis[x]:=⊥;
                     infl[x]:=∅;
                   for all x ⊒ t in S do
                     for all x' in FV(t) do
                       infl[x']:=infl[x'] ∪ { x ⊒ t};
          Step 2:  Iteration(updating W and Analysis)
                   while W ≠ empty do
                     ((x ⊒ t),W) := extract(W);
                     new:=eval(t,Analysis);
                     if Analysis[x] ⋣ new then
                       Analysis[x] := Analysis[x] ∪ new;
                       for all x' ⊒ t' in infl[x] do
                         W:=insert((x' ⊒ t'),W);
Using:    function eval(t,Analysis)
          return [t](Analysis)
          value empty
          function insert((x ⊒ t),W)
          return ···
          function extract(W)
          return ···
```

표 1: 할일하기 알고리즘(worklist algorithm)[2]

할일하기 알고리즘은 메모리 사용과 시간비용에 있어서 효율적이다. 정적 분석 방정식에서 소개한 프로그램 분석 방정식은 iteration 단계를 진행할때마다 $\vec{X}$을 모두 F에 적용시켜 계산하게 되는 부담이 있다. 할일하기 알고리즘은 이것을 해결하고자 변화된 값들을 찾아 그 값이 영향을 주는 부분을 worklist에 넣어 다음에 계산할때 그 부분이 적용되도록 한다. 즉, 값이 변하여 방정식을 다시 풀 부분만 골라서 계산하는 방식이다.

이에 대한 간단한 식은 다음과 같다.

$$
\begin{array}{rcl}
 & & (\text{k는 iteration 단계}) \\
\vec{X} & = & \langle x_1, \cdots, x_n \rangle \text{ , } J_k \in 2^{\{1,\cdots,n\}} \text{ 일때,} \\
\\
\vec{X}^0 & = & < \bot_1, \bot_2, \cdots, \bot_n > \\
\vec{X}^{k+1} & = & \begin{cases} F_i(\vec{X}^k) & \text{만약 } i \in J_{k+1} \text{ 라면} \\ \vec{X}^k & \text{그렇지않다면} \end{cases}
\end{array}
$$

iteration을 수행하기 위해서는 다음 할일이 worklist J에 속해 있으면 그 일을 수행한다. 다음 진행을 위해 worklist J에서 할일을 선택하는 식은 다음과 같다.

$$
\begin{array}{rcl}
J_1 & = & \{1\} \\
J_{k+1} & = & \{i \mid X_i \; influenced \; by \; X_j, j \in J_k\}
\end{array}
$$

1.1.3 넓히기

프로그램 분석은 유한한 시간내의 모든 프로그램을 분석해야 한다. 대상 프로그램이 유한한 시간내에 실행되는지 혹은 대상 프로그램이 무한히 반복되는 프로그램인지, 그러한 프로그램 속성에 관계없이 프로그램 분석은 종료해야 한다. 대상 프로그램의 의미에 따라서 분석기도 같이 무한히 실행된다면, 그 프로그램이 어떠한 성질을 갖는다라고 결론짓을 수 없다.

프로그램 분석 방정식의 식중에서 의미공간 D의 높이가 유한하다면 다음의 식을 그대로 계산하면 되지만, 만약 의미공간이 무한하다면

$$\bigsqcup_{i \in \mathbb{N}} F_C^i(\bot_1, \bot_2, \cdots, \bot_n)$$

의 계산은 끝나지 않게 될 것이다. 이때에는

$$\bigsqcup_{i \in \mathbb{N}} F_C^i(\bot_1, \bot_2, \cdots, \bot_n) \subseteq \lim_{i \in \mathbb{N}} (X_i)$$

유한한 체인 $\{X_i\}_i$를 찾는다.

이때, F_C가 단조(monotonic)함수이면, 그러한 체인 $\{X_i\}_i$ 은 F_C에 넓히기($\triangledown$)를 적용하여 다음과 같이 계산된다.

$$
\begin{array}{rcl}
X_0 & = & \bot \\
X_{i+1} & = & \begin{cases} X_i & \text{만약 } F_C(X_i) \subseteq X_i \text{ 이면} \\ X_i \triangledown F_C(X_i) & \text{그렇지 않다면} \end{cases}
\end{array}
$$

프로그램 분석에서 구하고자 하는

$$\bigsqcup_{i \in \mathbb{N}} F_C^i(\perp_1, \perp_2, \cdots, \perp_n)$$

은 위에서 정의한 넓히기를 적용하게 된다.

논문에서 논의되는 넓히기는 구간범위(interval domain)에 대해서 [-∞, a] 혹은 [a, ∞] (a는 임의의 정수)로 작동하는 넓히기에 대해서 논하겠다. 이에 대한 넓히기 정의는 다음과 같다.

$$[a, b] \nabla [c, d] = [e, f]$$

$$e = \begin{cases} a & \text{만약 } a \leq c\text{라면} \\ -\infty & \text{그렇지않다면} \end{cases}$$

$$f = \begin{cases} b & \text{만약 } b \geq d\text{라면} \\ \infty & \text{그렇지않다면} \end{cases}$$

1.2 넓히기의 정확도를 떨어뜨리는 할일하기 알고리즘 문제

프로그램 분석기중 버퍼오버런 오류를 찾는 아이락은 프로그램 분석의 여러 기술이 쓰인 분석기이다. 많은 프로그램의 분석 기술중 넓히기와 할일하기 알고리즘을 이용하여 iteration 과정을 계산하는 부분이 존재한다. 이 분석기에서 넓히기와 할일하기 알고리즘이 연결된 부분의 알고리즘을 따로 써보면, 표 2와 같다.

표 2에 기술되어 있는 알고리즘을 요약하면, 다음과 같다. worklist에서 할일 j을 선택한다. 선택된 일은 실행함수(F)에 적용한다. 이 결과을 넓히기 적용하여 값의 변화가 생기면 넓히기 포인트로 값을 바꾸게 된다. 실행함수를 적용하여 값의 변화가 생길 경우, 그 적용된 일 j에 영향을 받는 식들을 다시 worklist에 넣는다. 이러한 과정을 worklist에 할일이 남아있지 않을 때까지 반복하는 알고리즘이다.

프로그램 분석중에 넓히기와 할일하기 알고리즘을 사용할 때 정확도를 떨어뜨리는 문제를 살펴보기 위해 다음과 같은 간단한 반복문을 고려해보자.

X_0		$X_0 = \perp$
	i=0;	
X_1		$X_1 = X_0\{i \to [0,0]\}$
	while(i< 5){	
X_2		$X_2 = X_1 \cup X_3$
	i=1;	
X_3		$X_3 = X_2\{i \to [1,1]\}$
	}	

위의 프로그램에 대한 그래프는 그림 1과 같다. 이 프로그램은 i를 초기값 0으로 설정하고 i의 값이 5보다 작을 때까지 반복문 안의 수행문을 실행시키는 프로그램이다. 이 프로그램은 반목

$W \leftarrow \{1, \cdots, n\}$
$k \leftarrow 0$
$\forall X_i^0 \leftarrow \bot$
repeat
  $j \leftarrow choose(W)$
  $X_j^{k+1} \leftarrow X_j^k \triangledown F_j(\vec{X}^k)$
  if $X_j^{k+1} \not\subseteq X_j^k$, then insert $\{i \mid X_i \text{ depends on } X_j\}$ in W
  k++
until $W = \emptyset$

표 2: 넓히기와 할일하기 알고리즘을 조합한 알고리즘

문의 수행문이 i의 값을 계속 1로 설정하고, 설정된 i는 계속 반복문의 조건에 맞으므로 무한수행되는 프로그램이다.

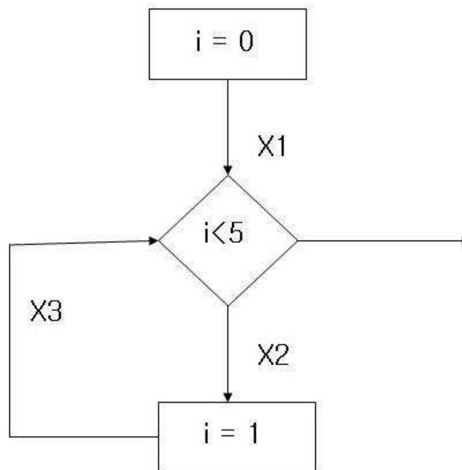

그림 1: 예제 프로그램에 대한 흐름 그래프

위의 식을 할일하기 알고리즘을 사용하여 iteration을 살펴보면, 그 과정은 표 3과 같다. 과정을 기술해보면, 현재의 iteration X_i^k에서 다음 iteration X_i^{k+1}을 풀기 위해서 어느 방정식을 풀어야 할지 알려주는 i를 worklist J_k 집합에서 선택하는데, 이 i는 이미 workist J_K 안에 있는 상태이다. worklist에 들어있는 i을 뽑아 다음 iteration 단계를 계산하게 되고, 또 이 계산에 영향을 받는 방정식 j을 다시 worklist J에 넣으므로써 다음의 iteration 단계를 수행한다. 값의 변화가 없을 때까지 반복하게 된다.

무한으로 분석되는 수행을 막기 위해 넓히기가 작동되어, 분석 결과 X_2 지점의 값은 $[0, \infty]$로 분석되는 것을 알 수 있다. 예제 프로그램은 값이 증가하는 프로그램이 아니라 임의의 상수값으로 설정하는 프로그램임에도 불구하고 값의 증가로 인해 넓히기가 수행되어 값이 터지는 것

X_i	0	1	2	3	4	5
X_1	$\perp$	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]
X_2	$\perp$	$\perp$	[0,0]	[0,0]	[0,∞]	[0,∞]
X_3	$\perp$	$\perp$	$\perp$	[1,1]	[1,1]	[1,1]
worklist	$J_0 = \{1\}$	$J_1 = \{2\}$	$J_2 = \{3\}$	$J_3 = \{2\}$	$J_4 = \{3\}$	$\emptyset$

표 3: 할일하기 알고리즘을 사용한 iteration

X_i	0	1	2	3
X_1	$\perp$	[0,0]	[0,0]	[0,0]
X_2	$\perp$	$\perp$	[0,1]	[0,1]
X_3	$\perp$	[1,1]	[1,1]	[1,1]

표 4: 벡터 iteration

을 알 수 있다.

그렇다면 보다 향상된 정확도로 분석을 수행하기 위해 넓히기와 할일하기 알고리즘을 조합한 분석이 어디에서 정확도를 떨어뜨리는지 알아보기 위해, 위의 예제 프로그램을 프로그램 분석 이론으로 iteration 과정을 살펴보면 표 4와 같다.

즉 이론적으로 방정식을 풀면 넓히기를 적용해도 [0,∞]로 값이 바뀌는 경우가 발생하지 않는다. 방정식이 X_2의 식이 X_1과 X_3로 나누어져서 계산할 때, 넓히기와 할일하기 알고리즘을 적용했을 때에는 X_1이 방정식이 계산되자마자 그 값이 영향을 주는 X_2로 전파되어 X_2 계산이 수행된다. 이때의 X_2의 계산은 X_3의 값이 구해지지 않은 상태에서 식을 계산한다. X_3의 값이 알려지지 않은채 X_1만의 값의 작은 변화만으로 넓히기가 수행되어 정확도를 떨어뜨리는 결과를 발생시킨다. 하지만, 이론적으로 X_2의 방정식은 X_1과 X_3의 값이 정해진 상태에서 계산하여야 정확한 분석을 수행할 수 있다. X_1과 X_3의 조인된 값에 넓히기가 적용되어야 위의 표 1.4와 같이 정확한 분석을 수행할 수 있다.

그림 2 는 넓히기의 수행시점에 따라 넓히기 작동없이 고정점에 도달하거나 혹은 넓히기가 작동하여 넓히기 포인트까지 도달하는 두 예를 보여주고 있다. 즉 iteration 단계를 촘촘히 계산하는 알고리즘은 고정점에 도달할 수 있는 값인데도 불구하고 넓히기가 수행되어 정확도를 떨어뜨린다. 이를 큰 폭으로 계산되는 알고리즘으로 수정한다면 고정점에 도달하여 넓히기를 수행하지 않아도 된다.

즉 위 예제 프로그램에 대해서 설명하면, 할일하기 알고리즘과 넓히기가 적용된 분석기에서는 고정점에 도달하는 프로그램에 대해서 고정점에 도달하지 않고 넓히기가 동작하여 넓히기 포인트까지 도달하는 분석을 수행하였다. 그러나 프로그램 분석 이론을 통한 프로그램 분석은 분석대상이 넓히기가 동작할 필요없이 바로 고정점에 도달하여 정확한 값을 분석했다.

2 해결책

프로그램 분석은 단순히 할일하기 알고리즘을 적용해서는 반복문의 무한수행을 제어하는 넓히기의 정확도를 보장할 수 없다.

```
X_i^0 ← ⊥
k ← 0
A ← 1 :: 2 :: 3 ··· :: n
W ← ϵ
repeat
   (j,A,W) ← choose(A,W)
   X_j^{k+1} ← X_j^k ∇ F_j(X⃗^k)
   if X_j^{k+1} ⊈ X_j^k , then add(i,A,W)
until (A = ϵ) ∩ (W = ϵ)
add :  ℤ × ℤ* × ℤ* → ℤ* × ℤ*
add(i,A,W) =
   foreach j ∈ influenced by (i)
   if loop-head(j)
      if j∉ W
          (A,W) := add(j,A,j::W)
   else
      (A,W) := (j::A,W)
   (A,W)
choose :  ℤ* × ℤ* → ℤ × ℤ* × ℤ*
choose(A,W) =
   case A
     i::A′ → (i,A′,W)
    | ϵ → case W
             i::W′ → (i,ϵ,W′)
           | ϵ → (-1,ϵ,ϵ )
```

표 5: 할일하기 적용하면서 계산을 늦추는 알고리즘

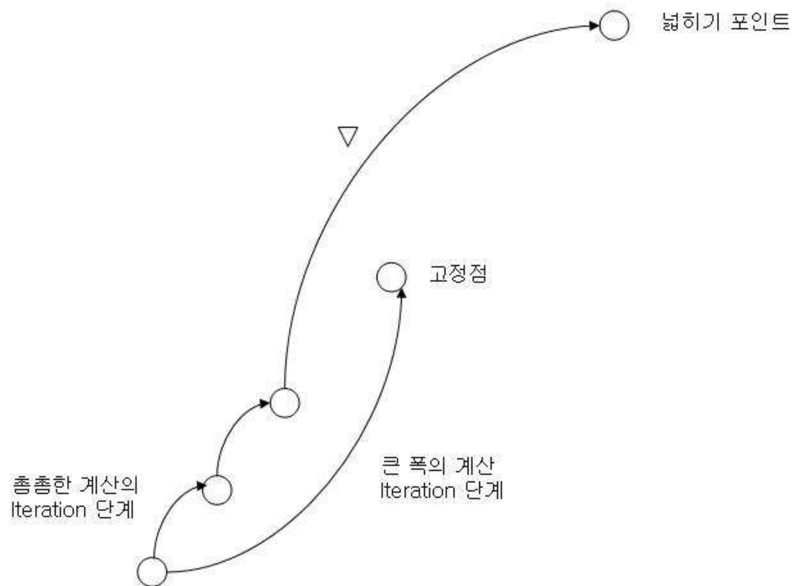

그림 2: iteration 단계를 촘촘히 계산한 결과와 큰폭으로 계산한 결과 비교

2.1 할일하기 알고리즘의 수정

앞에서 살펴본 바와 같이, 아이락에서 할일하기 알고리즘과 넓히기가 수행되는 과정에서 그 정확도를 저해하는 결과를 볼 수 있었다.

반복문에서 합쳐지는 두 식이 필요이상으로 할일하기 알고리즘에 의해 촘촘히 계산되어, iteration 단계 동안 이 촘촘한 계산이 넓히기를 작동시켜 부정확한 결과를 낳게했다.

이를 해결하기 위해 iteration 단계 동안 할일하기 알고리즘에 의해 변하는 값에 대해서 합쳐지는 두 식이 모두 계산될 때까지 계산을 미뤄 한쪽 식의 변한 값이 그 때의 iteration 방정식 도중에 쓰이지 않도록 잠시 미뤄두는 알고리즘을 소개한다.

할일하기 알고리즘에 넓히기를 적용할 때 정확도가 저해되는 점을 개선하도록 할일하기 알고리즘을 다음에 대해 수정한다.

- 할일하기 알고리즘에서 할일을 선택하는 방법
- 할일하기 알고리즘에서 변하는 식이 다른 수식에 영향을 미칠때 이를 처리하는 방법

이 두가지 수정된 부분이 표 5 에 반영되고 있다. 표 5 에 소개된 알고리즘에 대해서 좀 더 자세히 설명하면 다음과 같다.할일하기 알고리즘은 active-worklist(A)와 wait-worklist(W)로 나누어진다. 프로그램의 의미에 따라 프로그램의 처음 방정식이 A-worklist에 저장된다. 이후 프로그램 의미대로 분석을 수행한 후, 값이 변하면 add 함수가 변화된 값에 관계되는 j를 찾는다. 이러한 j가 반복문의 해드이면서 W-worklist에 저장되어 있는 것이 아니라면, j를 W-worklist에 저장하고 반복문에서 수행되는 방정식을 전부 A-worklist에 저장한다. 여기서 W-worklist에 저장되어 있는지 아닌지 확인하는 것은 조건에 맞는 방정식을 반복해서 W-worklist에 계속 넣는 무한분석을 막기 위한 것이다. 반복문에서 수행되는 방정식을 모두 계산하면 W-worklist에서 반복문의 해드 방정식을 계산하게 된다. 결국, 반복문에 흘러들어오는 식과 반복문에서 실행되는 식들을 계산한 후 반복문의 해드방정식을 계산하는 것이다.

X_i	0	1	2	3	4
X_1	$\perp$	[0,0]	[0,0]	[0,0]	[0,0]
X_2	$\perp$	$\perp$	$\perp$	[0,1]	[0,1]
X_3	$\perp$	$\perp$	[1,1]	[1,1]	[1,1]
A-worklist	1	3	ϵ	3	ϵ
W-worklist	ϵ	2	2	ϵ	ϵ

표 6: 넓히기로 인한 정확도 저하를 보완하는 알고리즘을 적용하여 iteration 방정식을 계산하는 과정

종류	알고리즘 수정전		알고리즘 수정후		줄 수
	알람 수	시간(초)	알람수	시간(초)	
grep-2.5.1	63	1177.99	63	2186.47	9,297
gzip-1.2.4	127	160.94	122	194.08	7,327
openssh-4.0	294	2003.29	273	2594.01	1,492,986
tar-1.13	63	1255.74	63	1440.22	20,258
tcl-8.5a3	12	55.75	12	70.19	12,359
sed-4.0.8	54	8433.55	37	8833.85	6,053

표 7: 실험결과

이 알고리즘을 적용하여 각 iteration이 어떻게 되는지 나타내는 것이 표 6 이다. 표 6 에서와 같이 매 iteration에서 나타나는 결과는 프로그램 분석 이론을 적용한 iteration 결과와 같다. 즉, 이 결과는 넓히기 수행을 막으므로 X_2 에는 [0,1]라는 정확한 값을 갖도록 하여 분석기의 정확도를 높여준다.

3 실험

본 논문에서 제안된, 넓히기로 인한 정확도 저하를 보완하는 할일하기 알고리즘의 성능을 실험하기 위하여 제안된 방법을 메모리 영역을 벗어나서 접근하는 C 프로그램 오류(buffer overrun error)의 위치를 미리 모두 자동으로 찾아주는 아이락에 대해서 적용하였다.

아이락의 알람에는 프로그램의 실제오류 알람(true alarm) 뿐만 아니라 프로그램의 거짓알람(false alarm)이 동시에 존재한다. 아이락은 정적 프로그램 분석 기술을 이용한 프로그램 분석 도구이다. 프로그램 정적 분석 기술은 프로그램이 실행중에 가지는 성질을 실행전에 자동으로 안전하게 어림잡는 일반적인 방법이다. 즉 프로그램을 실행시키지 않고 자동으로 프로그램이 실행중에 가지는 모든 상황을 빠뜨림없이 고려한다는 뜻이다. 따라서 프로그램 정적 분석 기술을 이용한 분석에는 안전하게 어림잡는 기술이 사용되어 거짓알람(false alarm)이 포함되기도 한다. 이 거짓알람의 수에 따라 분석기의 정확도가 판별된다.

실험은 할일하기 알고리즘을 수정하여 아이락에 적용한 결과와 대조군으로 할일하기 알고리즘으로 분석하는 아이락의 결과를 비교하였다. 프로그램 분석을 위한 기계로는 4GB 주 메모리를 가진 Pentium 4 3GHz CPU 컴퓨터를 사용하였다.

종류		알고리즘 수정전		알고리즘 수정후		줄 수
		알람 수	시간(초)	알람수	시간(초)	
gzip-1.2.4	unlzh.c	23	0.66	18	1.19	199
openssh-4.0	auth1.c	8	1.68	4	2.27	9,619
	channels.c	16	95.44	5	208.23	83,671
	cipher.c	12	1.90	11	3.96	12,122
	sshconnect2.c	14	25.22	9	27.54	40,248
sed-4.0.8	regex.c	54	8433.15	37	8833.00	250

표 8: 세부 결과

표 7 은 gnu 소프트웨어중 grep, gzip, openssh, tar, tcl, sed에 대해서 실험한 결과를 보여주고 있다. 또한 표 8 은 실험결과의 알람수중에서 알람수의 차이를 발생시킨 구체적인 소스에 대한 세부적인 자료이다. 위 실험은 아이락의 cf 옵션으로 프로그램을 분석한 결과이다. cf 옵션은 프로그램이 여러 파일로 나누어질때, 모든 파일들을 대상으로 분석을 하지 않고 각 파일마다 프로그램 분석을 하여 전체파일 대상으로 분석시 메모리 부족현상을 방지할 수 있다.

표 7 에 나타나듯이 넓히기로 인한 정확도 저하를 보완하는 할일하기 알고리즘을 적용한 아이락에서 정확도가 더 높은 결과가 나오고 있다. 수정된 알고리즘을 적용하기 전의 결과에서 openssh-4.0와 gzip-1.2.4 그리고 sed-4.0.8 에서는 알람수가 적게 나오는 것을 알 수 있다.

프로그램에 따라서 그 알람수의 차이는 다르게 나타난다. 논문에서 제시된 알고리즘을 사용했을 때, 넓히기 포인트(widening point)에 도달하기 전에 미리 고정점(fixpoint)에 도달한다면 이에 따라 알람수가 감소할 것이다. 하지만, 논문에서 제시된 알고리즘을 적용한 분석결과와 할일하기 알고리즘을 적용한 분석결과가 똑같이 나타난다면 그 알람수가 같을 것이다.

넓히기로 인한 정확도 저하를 보완하는 할일하기 알고리즘은 분석의 수행시간면에서는 기존의 할일하기 알고리즘에 비해 시간비용이 크다. 위의 실험결과에서 나타나듯이 프로그램마다 다르지만, 평균적으로 약 28% 떨어진다. 정확도는 향상되지만 프로그램의 수행시간은 오래 걸린다.

수정된 알고리즘의 수행시간이 늘어난 이유는 다음과 같다. 분석하고자 하는 어떠한 변수가 반복문에서 고정점에 도달하지 않고 계속 증가하는 변수를 갖는 프로그램을 고려해보자. 만약 기존의 알고리즘에서는 합쳐지는(join) 지점에서 한 식의 증가되는 결과만 나타나면 다음 계산을 하지않고 바로 넓히기 포인트로 값을 바꾸었지만, 정확도를 보완하는 할일하기 알고리즘에서는 합쳐지는 지점에서 한 식의 증가를 결정짓기 위해서 다음 식의 값이 어떻게 되는지 계산한 후 그 다음 그 두 식을 조인하여 전체식이 증가하는지 그렇지 않은지 판단하기 때문에 계산의 단계가 전에 비해서 많다. 그렇기 때문에 수정된 알고리즘은 시간비용이 크다.

다음의 간단한 예를 살펴보자.

X_i	0	1	2	3	4	5	6
X_1	$\perp$	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]
X_2	$\perp$	$\perp$	[0,0]	[0,0]	[0,∞]	[0,∞]	[0,∞]
X_3	$\perp$	$\perp$	$\perp$	[1,1]	[1,1]	[1,∞]	[1,∞]
worklist	$J_0 = \{1\}$	$J_1 = \{2\}$	$J_2 = \{3\}$	$J_3 = \{2\}$	$J_4 = \{3\}$	$J_5 = \{2\}$	$\emptyset$

표 9: 할일하기 알고리즘을 사용한 iteration 과정

X_i	0	1	2	3	4	5	6	7	8	9
X_1	$\perp$	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]
X_2	$\perp$	$\perp$	$\perp$	[0,0]	[0,0]	[0,0]	[0,∞]	[0,∞]	[0,∞]	[0,∞]
X_3	$\perp$	$\perp$	$\perp$	$\perp$	[1,1]	[1,1]	[1,1]	[1,∞]	[1,∞]	[1,∞]
A-worklist	1	3	ϵ	3	3	ϵ	3	3	ϵ	ϵ
W-worklist	ϵ	2	2	ϵ	2	2	ϵ	2	2	ϵ

표 10: 넓히기로 인한 정학도 저하를 보완하는 할일하기 알고리즘을 사용한 iteration 과정

X_0		$X_0 = \perp$
	i=0;	
X_1		$X_1 = X_0\{i \to [0,0]\}$
	while(i< 5){	
X_2		$X_2 = X_1 \cup X_3$
	i++;	
X_3		$X_3 = X_2\{i \to i + [1,1]\}$
	}	

이 예제를 할일하기 알고리즘과 넓히기로 인한 정확도 저하를 보완하는 할일하기 알고리즘에 적용하여 어떻게 itertation이 바뀌어지는지 살펴보자.

표 9 는 기존의 할일하기 알고리즘을 사용한 iteration 과정을 나타내고 있다. 또한 표 10 은 넓히기로 인한 정확도 저하를 보완하는 할일하기 알고리즘을 사용한 iteration 과정을 나타내고 있다. 이 두 알고리즘을 이용한 분석결과 모두 같은 범위의 값을 분석해내고 있다. 하지만, 두 알고리즘은 iteration 단계 차이를 드러내고 있는데 이러한 차이가 분석시간에 영향을 준 것이다.

4 결론

4.1 요약

할일하기 알고리즘에서는 두 식이 합쳐지는 점에서 하나의 식에 미리 값이 전달되면 전체 값이 계산되어 값이 커지거나 감소되어질 수 있다. 그러면 넓히기에 의해 값이 넓히기 포인트로 바뀌게 된다. 이는 원래 합쳐지는 전체 식이 고정점에 도달하지 않는 경우 넓히기가 빨리 실행되게 하여 조금이나마 시간적인 비용을 줄게하지만, 전체식이 만약 증가하지 않고 고정점에 도달

하는 경우라면 그 정확도를 저해하는 경우가 된다.

프로그램 분석 방정식을 프로그램 분석 이론으로 계산할 때와 같은 정확도를 갖기 위해서 반복문의 합쳐지는 점의 한쪽의 식들이 변한 값을 서로에게 전파시키지 않게 해야한다. 합쳐지는 양쪽의 식에서 한 식의 변화만으로 할일하기 알고리즘에 의해 다시 전체 식을 계산하게 하면 그 분석의 정확도를 떨어뜨릴 수가 있다.

할일하기 알고리즘에 적용된 넓히기는 분석의 정확도를 떨어뜨리는 경우를 발생시킨다. 이를 해결하기 위한 방법으로 합쳐지는 식에 대해서 값이 바뀐 하나의 식이 할일하기 알고리즘에 의해 바로 값이 전달되지 않도록 계산을 늦추는 방법을 제시한다. 합쳐지는 두식이 계산이 될때까지 기다리게 한 후, 넓히기가 작동할지 그렇지 않을지를 결정한다. 넓히기가 적동하지 않은 경우는 그 값이 고정점에 도달하는 경우이다. 이 알고리즘으로 분석의 정확도를 높여 보다 정확한 분석을 할 수 있다.

프로그램 방정식을 계산하는 세 가지 방법이 소개되었다. 하나는 프로그램 분석 이론으로 iteration을 계산하는 방법과 다른 하나는 할일하기 알고리즘을 적용하여 iteration을 계산하는 방법이다. 마지막으로 할일하기 알고리즘을 정확도 측면에서 개선시키는 방법이다. 각각에 대해서 정리하면 다음과 같다.

- 프로그램 분석이론으로 방정식 풀기

$$\vec{X}^{k+1} = F(\vec{X}^k)$$

($\vec{X}$:프로그램의 부품, F:프로그램의 실행함수, k:iteration 횟수)

- 할일하기 알고리즘을 적용하여 iteration 방정식 풀기

$$\vec{X}^{k+1} = \begin{cases} F_i(\vec{X}^k) & \text{만약 } i \in J_{k+1} \text{ 라면} \\ \vec{X}^k & \text{그렇지 않다면} \end{cases}$$

$$J_{k+1} = \{i \mid X_i \; influenced \; by \; X_j, j \in J_k\} \qquad J_1 = \{1\}$$

($\vec{X}$:프로그램의 부품, F:프로그램의 실행함수, k:iteration 횟수, i:프로그램 부품중 하나)

- 정확도를 향상시킨 할일하기 알고리즘을 적용하여 iteration 방정식 풀기

$$\vec{X}^{k+1} = \begin{cases} F_i(\vec{X}^k) & \text{만약 } i \in J_{k+1} \text{ 라면} \\ \vec{X}^k & \text{그렇지 않다면} \end{cases}$$

($\vec{X}$:프로그램의 부품, F:프로그램의 실행함수, k:iteration 횟수, i:프로그램 부품중 하나, J_{k+1}는 add함수)

이러한 세 방법의 차이의 결과는 표 2, 표 3, 표 6 에 잘 나타나 있다. 세 결과를 비교해보면 정확도를 향상시킨 할일하기 알고리즘이 유용하다. 첫번째 방법은 모든 방정식을 하나하나 계산해 나가는 방법으로 비용면에서 비효율적이다. 두번째 방법은 값이 고정점에 도달함에도 불구하고 넓히기를 수행하여 그 정확도를 저하시키는 경우가 발생한다. 마지막으로 정확도를 향

상시키는 할일하기 알고리즘을 이용한 방법은 할일하기 알고리즘을 분석의 정확도 면에서 향상시킨 방법이다. 이 수정된 할일하기 알고리즘은 할일하기 알고리즘을 유지하면서 프로그램 분석 이론을 적용한 계산보다 할일하기 알고리즘의 효율성을 갖고, 할일하기 알고리즘의 시간 비용 차원에서 효율은 떨어지지만, 프로그램 분석 이론의 정확성을 유지하는 방법이다.

참고 문헌

[1] ropas. ropas.snu.ac.kr/~kwang/4541.665A/06.

[2] Flemming Nielson, Hanne Riis Nielson and Chris Hankin. *Principles of Program Analysis* Springer, 1985.

[3] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified latiive model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 238-252, January 1977.

[4] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 269-282, 1979.

[5] Yungbum Jung and Jaehwang Kim and Jaeho Shin and Kwangkeun Yi. Taming False Alarms from a Domain-Unaware C Analyzer by a Bayesian Statistical Post Analysis. *Static Analysis: 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005. Proceedings.* , pages 203-217, September 2005.

[6] Francois Bourdoncle. Efficient chaotic iteration strategies with widenings. *In Conference on Formal Methods in Programming and their Applications, number 735 in LNCS,*, pages 128-141, 1993

[7] 정영범, 김재황, 신재호, 이광근. 자동 오류 검출을 위한 프로그램 분석기 - 아이락. 마이크로 소프트웨어, pages 178-186, June 2005.