논·문

증가분 계산에 기반한 고정점 계산을 위한 워크리스트 알고리즘*
(A Worklist Algorithm for Differential Fixpoint Evaluation)

안준선
한국항공대학교 전자정보통신컴퓨터공학부
jsahn@mail.hangkong.ac.kr

요 약

요약 해석에 기반한 프로그램 분석에서 분석 시간의 대부분은 고정점 계산을 위하여 소모된다. 이에 대하여, 고정점 계산을 위한 함수의 반복 적용 시에 함수의 적용으로 인한 새로운 증가분만을 계산함으로써 고정점 계산의 속도를 높이는 방법이 제안되었다. 본 논문에서는 이러한 증가분 계산에 기반한 고정점 계산 방법을 설명하고, 이를 위한 새로운 워크리스트(worklist) 알고리즘을 제시한다. 제시된 워크리스트 알고리즘은 각각의 계산 작업에 사용되는 변수값들의 증가분을 각각 따로 관리하지 않고도 워크리스트 내의 계산들의 스케쥴링을 자유롭게 할 수 있다는 장점이 있다. 제시된 알고리즘을 사용하여 요약 해석에 기반한 상수-이명 분석을 구현하였으며, 실제적인 프로그램들을 사용한 실험을 통하여 증가분에 기반한 함수값 계산과 적절한 워크리스트 스케쥴링을 사용하여 분석 시간을 절약할 수 있음을 보였다.

1. 서론

요약 해석(abstract interpretation)이란 래티스(lattices)로 표현되는 요약된 공간에서 프로그램을 수행해 봄으로써 프로그램의 성질을 파악하는 정적 분석 방법론을 말한다[1, 2, 3]. 요약 해석은 프로그램의 실제적인 의미(concrete semantics)와 요약된 의미(abstract semantics) 간의 관계를 안전성(soundness) 조건을 만족하는 추상화(abstraction) 및 실제화(concretization) 함수로 나타냄으로써, 프로그램 분석의 안전성을 보장할 수 있다. 또한 함수 포인터(pointers)나 고차 함수(higher-order functions)와 같은 구조를 가지고 있는 프로그래밍 언어의 분석을 일관된 방법으로 구현할 수 있으며, 요약의 정도를 조절함으로써 요구되는 분석 속도를 만족하도록 분석의 정밀도를 조절할 수 있는 장점을 가진다.

요약 해석에 기반한 프로그램 분석에서 분석 시간의 대부분은 고정점(fixpoints) 계산을 위하여 소모된다. 주어진 프로그램에 대한 정적 분석은 단조 증가 함수(monotonic increasing functions)의 형태로 표현될 수 있으며, 이 함수의 고정점이 안전(sound)한 분석 결과가 된다[4]. 함수의 고정점을 구하기 위해서는 해당 도메인의 가

*본 논문은 과학기술부·한국과학재단지정 "한국항공대학교 인터넷정보검색연구센터"의 연구비 지원으로 수행되었음.

장 작은 값(bottom)에서 시작하여 결과 값이 증가하지 않을 때까지 함수를 반복해서 적용하게 되는데, 일반적으로 프로그램의 크기가 커지고 요구되는 분석의 정확도가 높아질수록 요구되는 계산의 양이 증가한다.

고정점 계산의 속도를 높이기 위한 방법으로서 함수를 단순히 반복 적용하는 대신에 함수의 적용으로 인한 새로운 증가분만을 계산하여 이전 값과 합쳐주는 방법이 제시되었다[5, 6, 7, 8]. 이 방법에서는 이전 반복에서 수행한 계산이 다음 반복에서 중복되는 것을 피할 수 있으므로, 고정점 계산에 소요되는 전체 계산량을 줄일 수가 있고, 결과적으로 분석의 속도를 높일 수가 있다. 또한, 요약 해석을 위한 고정점 생성의 경우 대부분의 경우에 배분 법칙을 만족하지 않는데 이러한 경우에 증가분 계산 방법을 적용하여 효율적으로 고정점을 계산할 수 있는 방법이 제시 되었다[7, 8].

고정점 계산의 실제 구현을 위해서는 주어진 값의 증가에 대하여 모든 프로그램 부분에 대한 값을 다시 계산하는 대신에 증가한 부분의 값의 영향을 받는 부분만을 다시 계산하는 워크리스트 알고리즘이 사용된다. 그런데, 증가분 계산에 기반한 워크리스트 알고리즘은 최근의 계산값만이 아닌 최근의 정확한 증가분을 사용하여야 하기 때문에, 기존의 연구에서는 워크리스트의 스케쥴링으로 FIFO(First-In First-Out) 방식을 사용하므로써 이를 해결하였다[7, 8]. 그런데, 워크리스트의 스케쥴링은 고정점 계산의 효율성을 위하여 매우 중요하므로[11], 이러한 제약은 기존 증가분 계산에 기반한 고정점 계산 방법의 큰 취약점이 된다.

본 연구에서는 증가분 계산에 적합한 워크리스트 알고리즘을 제시한다. 제시된 알고리즘은 정확한 증가분을 사용하기 위한 제약을 만족하면서도 자유로운 워크리스트 스케쥴링이 가능하며, 이를 위한 추가적인 부담이 적다.

본 논문의 구성은 다음과 같다. 2절에서는 본 연구의 목적인 고정점 계산 문제를 제시하며 3절에서는 증가분에 기반한 고정점의 계산 방법을 설명한다. 4절에서는 새로운 워크리스트 알고리즘을 제시하며 5절에서 구현과 실험 결과를 설명하고 6절의 결론으로 맺는다.

2. 문제 정의

본 연구에서 대상으로 하는 문제는 완전 래티스(complete lattice)를 값 공간(value domain)으로 하는 다음과 같은 연립 방정식의 해를 구하는 것이다. 변수 x_i는 우리가 관심을 가지는 프로그램 각각의 부분의 성질을 나타내며, e_i는 다른 프로그램 부분들과 x_i가 나타내는 프로그램 부분의 성질간의 관계를 타나낸다.

$$\begin{aligned} x_1 &= e_1 \\ x_2 &= e_2 \\ &\cdots \\ x_n &= e_n \end{aligned}$$

x_i가 가지는 래티스 값들은 함수 값, 튜플값(tuple value), 집합값들을 포함하며, 이러한 값을 생성하는 요약식 e_i 는 다음과 같은 형태를 가진다.

$$\begin{aligned} Exp ::= \; & c \mid x \mid op\ e \mid e_1 \sqcup e_2 \mid (e_1, \ldots, e_n) \\ \mid \; & e.i \mid \{e\} \mid \texttt{mapjoin}\ (\lambda x.e')\ e \\ \mid \; & e_1[e_3/e_2] \mid \texttt{ap}\ e_1\ e_2 \\ \mid \; & \texttt{if}\ (e_0 \sqsubset e_1)\ e_2\ e_3 \\ \mid \; & \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} \end{aligned}$$

위의 형태들은 각각 래티스 도메인 상에서 상수, 변수, 기본 연산, 조인(join) 연산, 튜플식과 튜플식의 원소 추출, 집합값 생성, 집합의 각 원소에 대한 함수 적용, 함수값의 생성 및 갱신, 함수값의 적용, 조건문, 지역 변수의 사용을 나타낸다. 위와 같은 식의 형태는 요약 해석에 기반한 일반적인 프로그램의 정적 분석에서 생성되는 방정식을 표현할 수 있으며, 이와 비슷한 형태로 Z1이 제안되어 실제적인 프로그램 정적 분석의 기술에 사용되었다[9].

위와 같은 요약식에 대하여 계산 규칙 $Eval : Env \times Exp \rightarrow Val$이 주어질 수 있는데, 이는 변수들의 값을 저장한 환경과 요약식을 입력으로 받아 요약식의 값을 계산한다[8]. $Eval$ 함수는 요약식의 각각의 형태에 대하여 정의되는데, 예를 들어 상수, 변수, 튜플식과 지역 변수 선언의 계산 방법은 다음과 같이 정의될 수 있다.

$$Eval(E,\ c)\ =\ \mathbf{c}$$

$$Eval(E,\ x)\ =\ E(x)$$

$$\frac{Eval(E,\ e_i)\ =\ v_i\ (i = 1, 2)}{Eval(E,\ (e_1, e_2))\ =\ (v_1, v_2)}$$

$$\frac{\begin{array}{l} Eval(E,\ e)\ =\ v, \\ Eval(E + \{x \rightarrow v\},\ e')\ =\ v' \end{array}}{Eval(E,\ \texttt{let}\ x = e\ \texttt{in}\ e'\ \texttt{end})\ =\ v'}$$

상수는 그 자신의 값이 되고 변수의 값은 변수환경에 저장된 값이 된다. 또한 튜플식의 값은 각각의 부분식들의 값들의 튜플값이 되며, 지역변수 선언식에서는 새로 선언된 변수의 값을 환경에 저장하고 주어진 식을 계산하게 된다.

3. 증가분에 기반한 고정점 계산

2절에서 주어진 연립방정식 해는 $F(\overline{x}) = e$ (단, $\overline{x} = (x_1, \ldots, x_n)$, $e = (e_1, \ldots, e_n)$) 로 정의되는 함수 $F : Val^n \rightarrow Val^n$의 고정점(fixpoint)으로 주어진다. F가 단조 증가 함수일 경우에 F의 최소 고정점(least fixpoint)은 다음과 같은 알고리즘에 의하여 쉽게 구해질 수 있다.

$\overline{v} \leftarrow \overline{\bot}$
/* $\overline{v} = (v_1, \ldots, v_n)$, $\overline{\bot} = (\bot, \ldots, \bot)$ */
repeat {
  $\overline{v} \leftarrow F(\overline{v})$
} **until** ($\overline{v}$ does not change)

여기에서 $F(\overline{v})$의 계산은 $(Eval(E, e_1), \ldots, Eval(E, e_n))$에 의하여 수행될 수 있다 ($E = \{x_1 \rightarrow v_1, \ldots, x_n \rightarrow v_n\}$). 그런데 이러한 고정점 계산에는 함수 F를 계속해서 증가하는 결과에 적용하기 때문에 이전에 계산되었던 값에 대한 중복된 계산이 이루지게 된다. 또한, 일반적으로 증가된 래티스 원소에 대한 함수 적용은 더 많은 계산을 요구하므로 계산이 반복되면서 함수 적용에 더 많은 시간이 소요되게 된다. 그런데, 만약 주어진 함수 F에 대하여 다음과 같은 성질을 만족하는 증가분 계산 함수 F^δ를 찾아낼 수 있다면 고정점 계산에서 반복에 의한 중복된 계산을 줄일 수 있다.

$$\begin{array}{l} F(\overline{v} \sqcup \overline{v}^\delta) = F(\overline{v}) \sqcup F^\delta(\overline{v}, \overline{v}^\delta) \\ \quad \text{where } (v_1, \ldots, v_n) \sqcup (v_1^\delta, \ldots, v_n^\delta) \\ \qquad\qquad = (v_1 \sqcup v_1^\delta, \ldots, v_n \sqcup v_n^\delta) \end{array}$$

위와 같은 함수 F^δ를 사용한 고정점 생성 알고리즘은 다음과 같다.

$/* \ \overline{v} = (v_1, \ldots, v_n) \ */$
$/* \ \overline{\bot} = (\bot, \ldots, \bot) \ */$
$/* \ \overline{pv} = (pv_1, \ldots, pv_n) \ */$
$/* \ \overline{v}^\delta = (v_1^\delta, \ldots, v_n^\delta) \ */$
$\overline{pv} \leftarrow \overline{\bot}$
$\overline{v} \leftarrow F(\overline{\bot})$
$\overline{v}^\delta \leftarrow F(\overline{\bot})$
while $(\overline{v} \neq \overline{pv})$ {
  $\overline{v}^\delta \leftarrow F^\delta(\overline{pv}, \overline{v}^\delta)$
  $\overline{pv} \leftarrow \overline{v}$
  $\overline{v} \leftarrow \overline{pv} \sqcup \overline{v}^\delta$
}

이 알고리즘에서는 각각의 반복에서 이전의 결과와 증가분을 이용하여 새로운 증가분만을 구하여 이를 이전 결과와 합쳐 줌으로써 새로운 분석 결과를 생성한다. 이러한 방법을 통하여, 이전의 모든 결과에 반복적으로 F를 적용함으로써 생기는 중복된 계산을 피할 수 있게 된다.

$F^\delta(\overline{pv}, \overline{v}^\delta)$의 값은 증가분 계산 규칙 $Eval^\Delta : Env \times Env \times Exp \rightarrow Val$을 사용하여 $Eval^\Delta(\{x_1 \rightarrow pv_1, \ldots, x_n \rightarrow pv_n\}, \{x_1 \rightarrow v_1^\delta, \ldots, x_n \rightarrow v_n^\delta\}, e)$로 계산할 수 있다[8]. $Eval^\Delta$의 적용 $Eval^\Delta(E, E^\delta, e)$에서 E^δ는 변수 값들의 증가분을 가지는 변수 환경을 나타낸다. 증가분 계산 규칙 $Eval^\Delta$는 이전 변수 환경 E로부터 E^δ에 저장된 값만큼 변수 환경이 증가했을 경우 요약식 값의 증가분을 계산한다. 예를 들어, 상수, 변수, 튜플식, 지역 변수 선언식에 대한 증가분 계산 규칙은 다음과 같다.

$$Eval^\Delta(E, E^\delta, \ c) \ = \ \bot$$

$$Eval^\Delta(E, E^\delta, \ x) \ = \ E^\delta(x)$$

$$\frac{Eval^\Delta(E, E^\delta, \ e_i) = v_i \ (i = 1, 2)}{Eval^\Delta(E, E^\delta, \ (e_1, e_2)) = (v_1, v_2)}$$

$$\frac{\begin{array}{l} Eval(E, \ e) \ = \ v, \\ Eval^\Delta(E, E^\delta, \ e) \ = \ v' \\ Eval^\Delta(E + \{x \rightarrow v\}, E^\delta + \{x \rightarrow v'\}, e') = v'' \end{array}}{Eval^\Delta(E, E^\delta, \ \texttt{let} \ x = e \ \texttt{in} \ e' \ \texttt{end}) = v''}$$

상수식의 경우에는 환경에 값이 영향을 받지 않으므로 증가분은 $\bot$이 되며, 변수식의 증가분은 증가분 환경에 의하여 구해지고, 튜플식의 증가분은 부분식들의 값의 증가분에 의하여 구해진다. 지역 변수 선언에 대해서는 지역 변수의 값의 증가분을 증가분 저장 환경에 반영한 후에 결과식의 값을 계산하게 된다.

다음 정리는 주어진 증가분 계산 규칙 $Eval^\Delta$가 안전한 증가분을 계산함을 나타낸다.

정리 1 $\forall E, E^\delta \in Env, \ e \in Exp$, *if*

1. $Eval(E \sqcup E^\delta, e) \in Lattice$
2. $\forall x \in dom(E) \cup dom(E^\delta)$,
   $E(x) \in Lattice$ *and* $E^\delta(x) \in Lattice$,

then the following holds true.

$$Eval(E \sqcup E^\delta, e) \ = \ Eval(E, e) \sqcup Eval^\Delta(E, E^\delta, e) \quad \square$$

4. 워크리스트 알고리즘

2절과 3절에서 제시된 고정점 계산 알고리즘은 프로그램의 한 부분에 대한 분석 값이 변화하였을 경우에 다른 모든 프로그램 부분들에 대한 분

```
/* X[ ] : 변수 x_i들의 값이 저장된 변수 환경                          */
/* use[ ] : 변수 x[i]의 값을 사용하는 표현식의 인덱스들이 use[i]에 저장됨  */
/* worklist : 다시 계산되어야 하는 표현식들의 인덱스를 저장             */

worklist ← {1, 2, ..., n}        /* 모든 표현식 인덱스들의 집합으로 초기화 */
while (worklist ≠ φ) {
  i ← get_element(worklist)
  x ← Eval(X, e_i)    /* e_i 계산 중 X[j]가 사용되면 i가 use[j]에 추가됨 */
  if (x ≠ X[i]) {        /* 새로 계산된 X[i]의 값이 이전 값보다 클 경우 */
    X[i] ← x
    worklist ← append(worklist ∪ use[i])
  }
}
```

그림 1: 일반적인 워크리스트 알고리즘

석 결과를 다시 계산하게 되므로 실제적인 구현에는 적합하지 못하다. 그러므로 실제적인 구현에서는 새로 증가된 부분의 영향을 받는 부분만을 워크리스트에 저장하여 다시 계산하는 고정점 계산 알고리즘이 사용되어 왔으며, 이와 함께 워크리스트의 효율적인 스케쥴링을 위한 방법이 연구되어 왔다[10, 11].

그림 1은 함수의 반복 적용에 기반한 일반적인 워크리스트 알고리즘이다. $use[i]$는 요약식 e_i의 값을 사용하는 표현식들의 인덱스가 저장되며, e_i의 값이 증가할 경우에 $use[i]$에 있는 인덱스들이 워크리스트에 추가된다. 이와 같이 워크리스트에 값의 증가 가능성이 있는 식들을 저장하고, 워크리스트에 있는 식을 하나씩 계산함으로써 좀 더 효율적으로 고정점을 계산할 수 있다.

증가분 계산에 기반한 워크리스트 알고리즘에서는 $Eval$ 대신에 $Eval^{\delta}$를 사용하여 증가분을 계산하고 이를 이전 값과 합쳐줌으로써 각각의 표현식의 새로운 값을 구할 수 있다. 그런데, 증가분에 기반한 고정점 계산에 의한 워크리스트 알고리즘에서는 기존의 워크리스트 알고리즘과는 다른 추가적인 제약이 요구되는데, 이는 워크리스트로부터 계산할 식을 선택하는 순서에 관련한 것이다. 일반적인 워크리스트 알고리즘에서는 단순히 사용되는 변수들의 최근의 값을 사용하면 되므로 임의의 순서를 사용할 수 있지만, 증가분 계산에 기반한 알고리즘에서는 각각의 변수의 정확한 증가분이 사용되도록 하여야 한다.

예를 들어 e_1의 값이 e_2와 e_3의 계산에 사용되고($use[1] = \{2, 3\}$), e_1의 값이 d_1만큼 증가하여 e_2와 e_3가 워크리스트에 추가되었다고 하자. 그리고, 또 다른 요약식값의 증가에 의하여 e_1이 다시 워크리스트에 추가되었다고 가정하였을 때($worklist \supset \{1, 2, 3\}$) 다음과 같은 경우를 생각할 수 있다.

- 만약 e_2와 e_3가 먼저 계산된다면, 두 요약식의 계산 완료 후에 e_1의 최근 증가분은 $\perp$으로 설정되어, 증가분 d_1이 중복되어 사용되

지 않도록 한다.

- 만약 e_1이 먼저 계산되어 다시 d_2만큼 증가한다면, e_2와 e_3를 위한 e_1의 값의 증가분으로 $d_1 \sqcup d_2$가 저장되어야 한다.
- 만약 e_2만 계산되고 e_1이 e_3의 계산 전에 다시 계산되어 d_2만큼 증가한다면(이때 e_2는 다시 워크리스트에 추가된다.) e_2와 e_3의 계산을 위한 e_1의 최근 증가분은 각각 d_1과 $d_1 \sqcup d_2$가 된다.

위와 같은 워크리스트 내의 각 요약식의 계산을 위한 적절한 증가분 저장 문제에 대하여, 기존의 워크리스트 알고리즘을 기반으로 할 때 다음과 같은 해결책을 생각할 수 있다.

- 워크리스트로 중복된 원소를 가지지 않는 FIFO 큐(queue)를 사용한다.
- 워크리스트 내의 각각의 계산될 요약식에 대하여 사용되는 변수들의 환경을 각각 따로 저장한다.

첫번째 방법은 [7, 8]에서 사용된 방법이다. 큐를 사용함으로써 위의 예의 경우에 e_1이 다시 계산되기 이전에 항상 e_2와 e_3가 먼저 계산되는 것을 보장하게 되며(즉 위 세가지 경우 중 첫 번째 경우만 발생한다.), 이를 통하여 어떤 식 e_i의 값을 사용하는 모든 요약식들의 계산이 동일한 e_i의 증가분을 사용하도록 할 수 있다. 그러나, 워크리스트의 적절한 스케쥴링은 알고리즘의 효율적인 수행에 매우 중요하므로[11], 이를 FIFO로 제한하는 것을 적절하지 못하다.

두번째 방법은 [5, 6] 등에서 사용된 방법이다. 그러나 기존의 사용 사례의 경우에는 배분 법칙을 만족하는 경우에서 사용되었기 때문에, 증가분만 저장하면 되었고 각각의 증가분에 대한 계산의 순서도 자유롭게 할 수 있으므로, 효율적으로 구현이 가능하였다. 그러나, 요약 해석을 위한 고정점 생성의 경우 대부분 배분 법칙을 만족하지 않기 때문에, 어떤 식의 값이 증가하였을 경우에 관련된 식들을 위한 증가분을 각각 별도로 계산하여야 하므로, 워크리스트 내의 계산될 식에 대하여 변수 환경을 따로 따로 유지하는 것은 구현에 있어 많은 부담이 따른다.

본 연구에서는 위와 같은 단점을 해결하기 위하여 기존의 워크리스트 알고리즘과 다른 접근법을 사용하였다. 기존의 워크리스트 알고리즘은 어떤 식의 값 증가에 대하여 새로 계산되어야 할 식들을 워크리스트에 넣어 주는데 반하여, 본 연구의 알고리즘은 계산되어 값이 증가된 식을 워크리스트에 넣는다. 그리고 워크리스트에서 어떤 식이 선택되면 해당 식들을 사용하는 식들을 모두 계산하게 된다. 이렇게 함으로써 어떤 식의 값을 사용하는 모든 식들이 같은 증가분을 사용하여 계산을 수행하도록 보장할 수 있으므로, 환경을 각각 따로 저장하지 않으면서도 워크리스트의 스케쥴링을 자유롭게 할 수 있다.

그림 2는 새로운 워크리스트 알고리즘이다. 알고리즘에서 $X[i], dX[i], V[i]$는 각각 e_i의 이전 값, 새로운 증가분, 이전 값과 증가분을 조인한 새로운 값을 나타낸다. $use[i]$는 e_i의 값을 사용하는 식들의 집합으로서, e_i의 값이 증가되면 $use[i]$에 포함된 식들의 값은 새로 계산되어야 한다. *worklist*는 계산할 식들이 저장되는 곳으로 기존의 알고리즘들은 e_i의 값이 증가되면 $use[i]$에 있는 식들이 저장되었으나 본 알고리즘에서는 e_i가 증가하면 i를 저장하고 후에 워크리스트에서 저장된 i가 선택되면 $use[i]$에 포함되는 식들의 값이 모두 계산되게 된다. 이러한 방법을 통하여 부분적

```
/* V[i] : 최근에 계산된 x_i(= e_i)의 값                                    */
/* X[i] : 이전 x_i 값 (⊥으로 초기화 됨)                                    */
/* dX[i] : x_i의 가장 최근 증가분                                          */
/* use[i] : x_i를 사용하는 표현식들의 집합                                 */
/* timestamp[i] : 각각의 표현식의 가장 최근 계산 시간                      */
/* worklist : 최근의 계산에 의하여 값이 증가한 표현식들이 저장됨.          */

for (i = 1 to n){
   V[i] ← Eval(X, e_i)   /* e_i 계산 시 X[j] 가 사용되면 i가 use[j]에 추가됨 */
   dX[i] ← V[i]
}

worklist ← {1, 2, ..., n}
while (worklist ≠ φ) {
   i ← Extract(worklist)
   foreach (w ∈ use[i])
      if (timestamp[i] > timestamp[w]) {
         dV ← Eval^Δ(X, dX, e_w)
        /* Eval^Δ(X, dX, e_w) 계산 시 X[j]나 dX[j]가 사용되면 w가 use[j]에 추가됨 */
         New_V ← V[w] ⊔ dV
         if (New_V ≠ V[w]) {
            worklist ← worklist ∪ {w}
            dX[w] ← dX[w] ⊔ dV
            V[w] ← New_V
         }
         update(timestamp[w])
      }
   X[i] ← V[i]
   dX[i] ← ⊥
}
```

그림 2: 증가분 기반 고정점 계산을 위한 워크리스트 알고리즘

워크리스트 스케쥴링 방법	FIFO		LIFO	
사용 알고리즘 / 입력 프로그램	Naive	Diff	Naive	Diff
TIS(102[1],6028[2])	12.18[3]	9.22	1.46	1.37
amoeba(36,6062)	42.86	14.46	22.14	3.11
simplex(448,8739)	117.78	45.96	72.63	6.53
gauss(54,4710)	33.50	19.40	8.27	1.42
gauss1(34,2863)	9.26	11.51	0.17	0.23
wator(45,3467)	52.26	67.66	2.23	1.58

1: 프로그램 내 프로시져의 수
2: 프로그램 내 표현식의 수
3: 수행시간 (초)

표 1: 상수-이명 분석을 위한 고정점 계산

으로 계산순서의 제약을 가하면서 워크리스트 내의 요약식 선택의 순서를 자유롭게 할 수가 있게 된다. 또한 모든 식에 대하여 최근의 계산 시점을 *timestamp*[*i*]에 저장함으로써 이미 최신의 값들을 사용하여 계산된 경우에 중복된 계산이 수행되지 않도록 하였다.

5. 구현 및 실험

프로그램 분석기 생성 도구인 Z1[9]를 사용하여 수행된 구현의 전체적인 구조는 다음과 같다. 먼저 분석하고자 하는 목적에 맞는 요약된 의미를 프로그램 분석 개발자가 정의하게 되고, 이를 입력으로 받아 분석식 생성기와 요약 공간에 대한 라이브러리를 자동으로 생성하게 된다. 분석식 생성기는 분석할 프로그램을 중간언어인 MIL[12]의 형태로 입력 받아 모든 부분식 간의 관계를 나타내는 방정식을 자동으로 생성하게 되며, 고정점 생성 알고리즘을 사용하여 생성된 방정식의 해를 계산함으로써 프로그램의 분석 결과를 얻게 된다. MIL은 Fortran과 C를 위한 전단부가 제공되기 때문에 위와 같은 방법을 통해 C와 Fortran 프로그램의 정적 분석을 수행할 수 있다.

증가분의 계산에 기반한 고정점 계산 알고리즘의 성능을 실험하기 위하여 상수 분석 및 이명 분석을 동시에 수행하는 프로그램 분석을 구현하였고 이를 실제적인 프로그램에 적용해 보았다. 실험 결과는 표 1과 같다. 입력 프로그램들은 시뮬레이션과 수치 계산을 위한 C와 Fortran 프로그램들이며 괄호 안의 숫자는 각각 프로그램 내의 프로시져의 갯수와 표현식의 갯수를 나타낸다. "Naive"는 그림 1의 일반적인 워크리스트 알고리즘을 사용한 경우, "Diff"는 증가분 계산에 기반한 워크리스트 알고리즘을 사용한 경우의 고정점 계산 소요 시간을 나타내며, 본 연구에서 제시된 워크리스트 알고리즘은 임의의 스케쥴링 방법을 사용할 수 있으므로, FIFO와 LIFO(Last-In First-Out)의 두 가지 스케쥴링 방법을 사용하여 실험을 수행하였다. 프로그램 분석을 위한 기계로는 256MB 주 메모리를 가진 Pentium 3 700MHz PC를 사용하였다.

전체적으로 LIFO 스케쥴링을 적용한 경우에 고정점 계산의 속도가 빨랐으며, 이는 스케쥴링 방법이 워크리스트 알고리즘에 있어서 매우 중요함을 나타낸다. 예제 프로그램 중 simplex, amoeba, gauss, TIS 프로그램에서 속도의 증가가 나타났으며, 특히 처음 세 프로그램의 경우 LIFO 스케쥴링과 증가분 기반 계산을 함께 사용함으로써 주목할 만큼 분석 시간을 감소시킬 수 있었다.

그러나, wator의 FIFO 스케쥴링과 gauss1의 경우 분석시간이 증가하였는데, 이는 새로운 증가분을 이전의 결과와 합쳐주는 계산으로 인한 부담이 증가분만을 계산함으로써 얻어지는 계산양의 감소보다 더 컸기 때문으로 판단된다.

6. 결론

증가분에 기반한 고정점 계산 방법은 반복된 함수의 적용시에 이전 결과에 그대로 함수를 적용하는 것이 아니라, 인자값의 증가를 고려하여 결과의 증가분만을 계산하도록 함으로써 고정점 계산에 소요되는 계산의 양을 줄인다. 본 연구에서는 이러한 방법의 실용적인 사용을 위하여, 증가분에 기반한 고정점 계산에 적합한 워크리스트 알고리즘을 제시하였다. 제시한 방법을 실제적인 프로그램의 상수-이명 분석에 적용해 본 결과, 다양한 워크리스트 스케쥴링에 있어 증가분 계산에 기반한 고정점 계산이 효율적임을 확인할 수 있었으며, 또한 증가분 계산에 기반한 고정점 계산에 있어서도 워크리스트 스케쥴링이 계산 속도의 향상에 매우 중요함을 알 수 있었다.

향후 연구는 다음과 같다. 우선 속도의 저하를 보인 프로그램들을 분석하여, 제시된 알고리즘의 부담을 정확히 분석해야 하며, 이를 극복할 수 있는 방법을 찾아내는 것이 필요하다. 또한, 일부 입력 프로그램의 경우에 LIFO 스케쥴링을 사용함으로써 증가분 기반 고정점 계산의 속도 향상이 두드러지는 현상을 보였는데, 이와 관련하여 증가분 기반 고정점 계산에 적합한 워크리스트 스케쥴링 방법을 연구하는 것이 필요한 것으로 판단된다.

참고문헌

[1] Patrick Cousot and Radhia Cousot, "Abstract interpretation: a unified lattice model for static analysis of program by construction of approximation of fixpoints," In *4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[2] Patric Cousot and Radhia Cousot, "Systematic design of program analysis frameworks," In *6th ACM Symposium on Principles of Programming Languages*, pages 269–282, 1979.

[3] Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages. Computers and Their Applications*, Ellis Horwood, 1987.

[4] Kwangkeun Yi, "Yet another ensemble of abstract interpreter, higher-order data-flow equations, and model checking," Technical Memorandum ROPAS-2001-10, Research On Program Analysis System, KAIST, March 2001.

[5] Francois Bancilhon and Raghu Ramakrishnan, "An amateur's introduction to recur-

sive query processing strategies," In *ACM SIGMOD Conference on Management of Data*, pages 16-52, 1986.

[6] Christian Fecht and Helmut Seidl, "Propagating differences: an efficient new fixpoint algorithm for distributive constraint systems," In *Proceedings of European Symposium on Programming (ESOP)*, pages 90–104. LNCS 1381, Springer Verlag, 1998.

[7] Hyunjun Eo, Kwangkeun Yi, "An Improved Differential Fixpoint Iteration Method for Program Analysis," *Proceedings of The Third Asian Workshop on Programming Language and Systems*, Shanghai, China, November 2002.

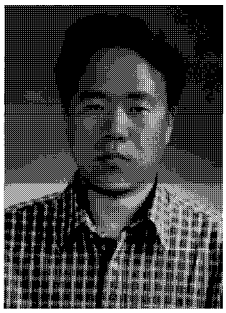
[8] Joonseon Ahn, "A Differential Evaluation of Fixpoint Iterations," *Proceedings of The Second Asian Workshop on Programming Language and Systems*, Daejon, Republic of Korea, Dec 2001.

[9] Kwangkeun Yi, *Automatic Generation and Management of Program Analyses*, Ph.D. Thesis, Report UIUCDCS-R-93-1828, CSRD, University of Illinois at Urbana-Champaign, 1993.

[10] Barbara G. Ryder and Marvin C. Paull, "Elimination algorithms for data flow analysis," *ACM Computing Surveys*, Vol.18, No.3, pages 277-316, 1986.

[11] Li-ling Chen, Luddy Harrison and Kwangkeun Yi, "Efficient computation of fixpoints that arise in complex program analysis," *Journal of Programming Languages*, Vol.3, No.1, pages 31–68, 1995.

[12] Williams Ludwell Harrison III and Zahira Ammarguellat, "A program's eye view of MIPRAC," In D. Gelernter, A. Nicolau and D. Padua, editors, *Languages and Compilers for Parallel Computing*. MIT Press, August 1992.



안준선
1988.3-1992.2 서울대학교 계산통계학과(학사)
1992.3-1994.2 KAIST 전산학과(석사)
1994.3-2000.8 KAIST 전자전산학과(박사)
2000.9-2001.8 KAIST 프로그램분석시스템-연구단(ROPAS) 연구원
2001.9-현재 한국항공대학교 전자정보통신-컴퓨터공학부 전임강사
관심분야는 컴파일러, 정적 분석, 병렬 언어 함수형 언어, 임베디드시스템, 정보 검색 등.