# 요약해석에서 증가분 계산에 기반한 고정점 생성 방법*

안준선[ν] 이광근
한국항공대학교 전자정보통신컴퓨터공학부[ν] 한국과학기술원 전산학전공 프로그램분석시스템연구단
{jsahn,kwang}@ropas.kaist.ac.kr

# Differential Evaluation of Fixpoints in Abstract Interpretation

Joonseon Ahn[ν] Kwangkeun Yi
School of Electronics, Telecommunications and Computer Engineering, Hankuk Aviation University
Research on Program Analysis System, Computer Science Dicision, KAIST

## 요 약

요약해석(abstract interpretation)이란 래티스(lattice)로 표현되는 요약된 공간에서 프로그램을 수행함으로써 프로그램의 성질을 분석하는 방법이다. 요약해석에서 프로그램의 분석 결과는, 정보가 없는 상태에서 시작하여 더 이상 정보의 증가가 없을 때까지 프로그램을 반복해서 수행함으로써 얻어지는 고정점(fixpoint)에 의하여 표현된다. 본 연구에서는 이러한 고정점 계산의 속도를 높이기 위하여 이전 반복의 계산 결과를 최대한 이용하는 방법을 제시한다. 그리고 제시된 방법을 상수 및 이명 분석의 구현에 적용하여 실제로 분석 속도가 증가함을 보였다.

## 1. 서론

정적 분석(static analysis)이란 프로그램 수행중의 관심있는 성질을 프로그램을 실행시키지 않고 미리 조사하는 것을 말한다. 이러한 정적 분석은 그 목적에 따라 상수 전달(constant propagation), 이명 분석(aliasing analysis), 예외상황 분석(exception analysis), 정적 분할(static slicing), 흐름 분석(control flow analysis) 등이 있으며, 프로그램의 최적화(optimization)나 안전성 증명을 위하여 사용된다. 우리가 관심을 갖는 이러한 정적 분석의 문제들은 프로그램을 수행하지 않고는 정확하게 알아낼 수 없는(undecidable) 문제들이 대부분이다. 그러므로, 대부분의 프로그램 분석 방법은 분석의 정확성을 어느 정도 희생하면서, 프로그램의 분석이나 검증에 유용한 수준의 안전한 정보를 생성한다.

요약해석[1,2,3]이란 래티스로 표현되는 요약된 공간에서 프로그램을 수행해 봄으로써 프로그램의 성질을 파악하는 방법을 말한다. 이 방법론에서는, 요약된 공간을 사용하고 우리가 관심을 가지는 정보가 항상 증가하므로 프로그램의 분석이 항상 유한시간 내에 끝나게 되며, 프로그램의 실제적인 의미(concrete semantics)와 요약된 의미(abstract semantics)간의 관계를 안전성 조건을 만족하는 추상화(abstraction) 및 실제화(concretization) 함수로 정의함으로써 프로그램 분석의 정확성을 보장할 수 있다. 또한 요약해석은 고차 프로그래밍 언어(higher-order programming languages)의 분석에 쉽게 적용할 수 있는 장점을 가지고 있으며, 다양한 프로그램 분석을 포함하는 강력한 프로그램 분석 방법임이 증명되었다.

그림 1은 간단한 플로우차트 프로그램에 대한 요약 분석의 예이다. 만약 우리가 각각의 에지에서 변수들의 값들이 가질

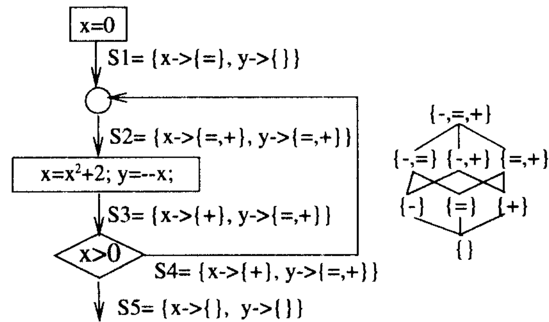


그림 1: 요약해석에 기반한 프로그램 분석의 예

수 있는 부호를 모두 알고 싶다고 할 때, 그림의 오른편과 같은 요약된 공간에서 수행함으로써 안전하면서도 어느 정도 유용한 정보를 얻을 수 있게 된다. 요약된 공간에서 $\{\}$은 아직 아무런 정보가 없음을 나타내고 $\{+,-,=\}$은 모든 부호를 가질 수 있음을 나타내며 다른 요약값들은 각각 그에 해당하는 부호의 값들을 가질 수 있음을 나타낸다. 즉, 어떤 에지에서 $x$값이 $\{-1,1,2\}$의 값을 가질 수 있다면 이는 요약된 공간에서 $\{-,+\}$로 나타내어지게 된다. 요약 수행시 처음에는 모든 변수의 값에 $\{\}$을 지정하고 수행하면서 가질수 있는 부호의 집합을 증가시키게 되며, 더 이상 분석결과가 증가하지 않는 고정점(fixpoint)에 도달하면 분석이 끝나게 된다. 그림 1의 예를 보면, 결합(junction) 노드를 수행한 다음의 $S2$의 값은 각각 반복하여 수행됨에 따라 $\{x \to \{\}, y \to \{\}\} \Longrightarrow \{x \to \{=\}, y \to \{\}\} \Longrightarrow \{x \to \{=,+\}, y \to \{=,+\}\}$ 와 같이 증가하면서 분석 결과를 생성하게 된다. 이러한 분석 결과를 보면, $S3$에서 $y$는 항상 양수인데 $\{=,+\}$로 분석된 것처럼 어느정도 부정확한 값을 생성하기는 하지만 가능한 모든

*본 연구는 과학기술부 창의적연구진흥사업의 지원을 받았음.

부호를 포함하는 안전한(sound) 분석 결과를 생성함을 볼 수 있다.

본 연구에서는 이러한 고정점 계산의 속도를 높임으로써 요약 해석에 기반한 프로그램 분석의 속도를 높이고자 한다. 이를 위하여 중첩된 계산을 최대한 피하고 증가값만을 새로운 계산에 활용하는 고정점 계산 방법을 제안한다. 이어지는 절에서는 중첩된 계산을 피하는 고정점 계산 알고리즘의 원리와, 증가값 계산을 주어진 분석에 대하여 자동으로 수행하기 위한 방법을 설명하고 실제 구현및 실험 결과를 제시한다.

## 2. 증가분에 기반한 고정점 계산 방법

### 2.1 기본 아이디어

요약해석에 기반한 프로그램의 분석은 우리가 관심을 갖고 있는 프로그램의 각각의 성질들 사이의 연립방정식으로 표현될 수 있다[4]. 예를 들어 그림 1에서 주어진 프로그램에 대한 분석은 $Si = F_i(S1, S2, S3, S4, S5)$로 나타내어지고($1 \leq i \leq 5$), $S = (S1, S2, S3, S4, S5)$, $F = (F_1, F_2, F_3, F_4, F_5)$라고 할때 분석의 결과는 연립방정식 $S = F(S)$ 의 해가 된다. 이때 만약 F가 단조 증가(monotonic increasing) 함수라면 해 S는 다음과 같이 구할 수 있다.($\perp = (\{\}, \{\}, \{\}, \{\}, \{\})$).

```
S = ⊥
do {
  S = F(S)
} until (S is not increasing)
```

그런데 이러한 고정점 계산에는 이전의 값을 이용하여 새로운 값을 생성하기 때문에 이전에 계산되었던 값에 대한 중첩된 계산이 이루지게 된다. 즉, 그림 1의 예를 보면, $S3$의 $x$ 값의 계산을 위해서는 $S2$의 $x$값이 사용되는데, $S2$의 $x$값이 $\{=\}$일때와 $\{=, +\}$일때 $S3$의 값을 구하게 된다. 이때 $x$값 $=$에 대한 계산이 중첩되어 이루어짐을 알 수 있다. 이러한 중첩된 계산을 없애고 두번째 $S3$의 $x$값 계산시에는 새로이 포함된 $+$에 대한 계산만을 수행하여 이전값과 합쳐줌으로써 전체적인 계산의 양을 줄일 수 있다.

만약 주어진 함수 $F$에 대하여 다음과 같은 성질을 만족하는 효율적인 함수 $F^\delta$를 찾아낼 수 있다면 고정점 계산에서 중첩된 계산을 줄일 수 있다.

$$F(v \sqcup v^\delta) = F(v) \sqcup F^\delta(v, v^\delta)$$

여기에서 $\sqcup$은 래티스 공간에서의 조인(join) 연산을 나타낸다. $F^\delta$는 $v^\delta$만큼의 입력값 증가에 의한 함수 $F$의 결과값의 증가분을 구하는 함수이다. 만약 $F$가 $\sqcup$연산에 대하여 분산법칙을 만족한다면 $F^\delta(v, v^\delta)$는 $F(v^\delta)$가 되며 중첩된 계산은 없어지게 된다. 또한 최악의 경우에는 $F^\delta(v, v^\delta)$는 $F(v \sqcup v^\delta)$로 주어질 수 있으며, 이 경우에는 중첩된 계산을 없애지 못하게 된다.

증가분을 계산하는 함수 $F^\delta$를 사용한 고정점 알고리즘은 다음과 같다.

```
pv = ⊥
v = dv = F(⊥)
while (v ≠ pv) {
  dv = F^δ(pv, dv)
  pv = v
  v = pv ⊔ dv
}
```

이 알고리즘에서는, 이전의 결과와 증가분에 의한 새로 더해진 결과값을 합쳐줌으로써, 이전의 모든 결과에 반복적으로 $F$를 적용함으로써 생기는 중첩된 계산을 피할 수 있게 된다. 이어지는 절에서는 주어진 $F$로부터 $F^\delta$를 찾아내는 방법을 제시한다.

### 2.2 증가분 계산 함수 $F^\delta$의 생성

본 연구에서는 $F$로부터 $F^\delta$를 직접 생성하는 대신에 $F$의 의미론(semantics)에 기반하여 $F^\delta$의 값을 구하는 방법을 사용한다.

먼저 함수 $F$의 형태를 정의한다. 함수 $F$는 분석결과를 나타내는 래티스의 원소들을 입력으로 받아서 다시 래티스의 원소를 생성하는 함수로서 $F(x_1, \ldots, x_n) = e$로 정의되며 이때 $e$는 $x_1, \ldots, x_n$를 포함하는 래티스식으로서 다음과 같은 형태를 갖는다.

$$\begin{array}{lcl} e \in Exp & ::= & c \mid x \mid (e_1, e_2) \mid e.i \mid e_1 \sqcup e_2 \mid \bigsqcup e \\ & & \mid \textbf{if } (e_0 \leq e_1)\ e_2\ e_3 \mid f\ e \\ & & \mid \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} \\ & & \mid e_1[e_2/e_3] \mid \textbf{ap } e_1\ e_2 \mid \textbf{map } \lambda x.e'\ e \end{array}$$

$c$는 상수이며, $x$는 변수, $e_1, e_2$)는 튜플(tuple), $e_i$는 튜플 원소의 선택, $e_1 \sqcup e_2$는 조인, $\bigsqcup e$는 집합 $e$의 원소들을 모두 조인한 것, $e_1[e_2/e_3]$는 함수 래티스의 원소 $e_1$의 $e_3$에 대한 값을 $e_2$로 지정하는 것을 나타내며, ap $e_1$ $e_2$는 함수 래티스의 적용을 나타낸다. 그외에도 조건문, 지역변수 선언과 함수를 집합의 각각의 원소에 적용하는 map 연산을 포함하고 있다.

이렇게 정의된 함수 $F$를 래티스값 $(v_1, \ldots, v_n)$에 적용한 결과는 함수의 몸체 $e$를 변수 환경 $E = [x_1 \to v_1, x_2 \to v_2, \ldots, x_n \to v_n]$에서 계산한 결과가 되며 이를 $Eval(E, e)$로 표기하기로 한다. $Eval$ 함수는 $e$의 각각의 형태에 대하여 귀납적으로 정의될 수 있는데 예를 들어 상수, 변수와 지역변수 선언의 계산 방법은 다음과 같이 정의될 수 있다.

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

$$\frac{Eval(E,\ e)\ =\ v, \quad Eval(E + \{x \to v\},\ e')\ =\ v'}{Eval(E,\ \textbf{let } x = e \textbf{ in } e' \textbf{ end})\ =\ v'}$$

상수는 그 자신의 값이 되고 변수의 값은 변수환경에 저장된 값이 된다. 또한 지역변수 선언식에서는 새로 선언된 변수의 값을 환경에 저장하고 주어진 식을 계산하게 된다.

이와 같이 주어진 $Eval$ 함수에 대하여 만약 변수 환경의

값들이 $x_1, \ldots, x_n$에 대하여 각각 $v_1, \ldots, v_n$에서 $v_1^\delta, \ldots, v_n^\delta$ 만큼 증가할 경우, 증가한 값에 의한 래티스식 $e$의 계산값의 증가분은 $Eval^\delta(E, E^\delta, e)$로 표시하기로 한다. 여기에서 $E^\delta$는 변수값들의 증가분을 나타내는 변수 환경으로서 $[x_1 \rightarrow v_1^\delta, x_2 \rightarrow v_2^\delta, \ldots, x_n \rightarrow v_n^\delta]$가 된다. 이러한 $Eval^\Delta$는 주어진 $Eval$의 의미에 기반하여 적절히 정의될 수 있는데, 예를 들어 상수, 변수와 지역변수 선언의 경우 다음과 같이 정의될 수 있다.

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

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

이와 같이 적절히 $Eval$과 $Eval^\Delta$를 정의할 경우 다음과 성질을 증명할 수가 있는데, 본 논문에서는 $Eval$과 $Eval^\Delta$의 모든 형태에 대한 정의와 증명은 지면관계상 생략하기로 한다.

**Theorem 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.*

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

위의 성질에 의해, $F(x_1, \ldots, x_n) = e$로 정의된 함수 $F$와 주어진 래티스값 $\bar{v} = (v_1, \ldots, v_n)$, $\bar{v}^\delta = (v_1^\delta, \ldots, v_n^\delta)$에 대하여 $F(\bar{v} \sqcup \bar{v}^\delta) = F(\bar{v}) \sqcup F^\delta(\bar{v}, \bar{v}^\delta)$를 만족하는 $F^\delta(\bar{v}, \bar{v}^\delta)$의 값은 $Eval^\Delta([x_1 \rightarrow v_1, \ldots, x_n \rightarrow v_n], [x_1 \rightarrow v_1^\delta, \ldots, x_n \rightarrow v_n^\delta], e)$로 계산될 수 있다.

## 3. 구현 및 실험

증가분의 계산에 기반한 고정점 계산 알고리즘의 성능을 실험하기 위하여 상수 분석 및 이명 분석을 동시에 수행하는 프로그램 분석을 요약해석에 기반한 프로그램 분석도구인 Z를 사용하여 구현하였다[5]. 그 실험 결과는 표 1과 같다. 괄호 안의 숫자는 각각 프로그램 내의 프로시저의 갯수와 표현식의 갯수를 나타낸다. non-incremental은 기존의 방법을 사용한 경우, incremental은 증가분 계산에 기반한 방법을 사용한 경우의 분석 소요 시간을 나타낸다.

simplex, amoeba, gauss 프로그램의 경우 어느정도 주목할 만한 분석 시간의 감소가 있었다. 그러나 wator, gauss1 프로그램의 경우 분석시간이 증가하였는데, 이는 최종결과를 생성하기 위하여 각각의 증가분을 합쳐주는 계산으로 인한 부담(overhead)이 증가분만을 계산함으로써 얻어지는 계산양의 감소보다 더 컸기 때문으로 판단된다.

표 1. 상수-이명 분석의 분석 소요 시간 (단위:초)

| program | incremental | non-incremental |
|---|---|---|
| simplex (44,8739) | 42.03 | 122.31 |
| amoeba (36,6062) | 18.63 | 48.47 |
| gauss (54,4710) | 12.38 | 28.18 |
| wator (45,3467) | 31.77 | 30.72 |
| gauss1 (34,1863) | 6.35 | 5.24 |

## 4. 결론

본 연구에서는 요약해석에 기반한 프로그램 분석의 속도를 높이기 위하여 증가분에 기반한 고정점 방법을 제시하였다. 그리고 주어진 방법이 프로그램 분석의 속도를 높일 수 있음을 보였다.

향후 연구는 다음과 같다. 일부 속도의 증가와 저하를 보인 프로그램들을 비교 분석하여 어떤 경우에 새로운 알고리즘이 효과적으로 적용될 수 있는지를 규명해야 한다. 또한 상수-이명 분석 뿐 아니라 다른 다양한 분석에 대한 실험도 더 필요할 것으로 판단된다.

## 참고 문헌

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

[2] Neil Jones and Alan Mycroft. Data Flow Analysis of Applicative Programs Using Minimal Function Graphs. In *13th ACM Symposium on Principles of Programming Languages*, pages 296–306, 1986.

[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] Kwangkeun Yi *Automatic Generation and Management of Program Analyses* Ph.D. Thesis, Report UIUCDCS-R-93-1828.

[6] 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.