증가분 계산에 기반한 고정점 생성 방법*
(A Fixpoint Iteration Method based on Increment Evaluation)

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

요 약

정적 분석의 속도를 향상시키기 위하여, 증가분(increment)만을 계산하여 고정점(fixpoints)을 효율적으로 구하기 위한 방법을 제시한다. 일반적인 고정점 계산 반복문(fixpoint iterations)에서는 단조증가(monotonic increasing) 성질을 갖는 함수를 함수의 결과가 증가하지 않을 때까지 반복 적용하기 때문에, 반복 간에 중복되는 계산을 수행하게 된다. 본 연구에서는 이전 계산 결과에 대한 증가분만을 계산하여 각각의 반복에서의 계산량을 줄이고자 하였다. 제시된 방법의 실용성을 검증하기 위하여, 요약 해석에 기반한 프로그램 분석기 생성 도구인 Z1을 사용하여 상수 전달 및 이명 분석을 위한 프로그램 분석기를 구현하고, 이를 실제적인 프로그램들에 적용하여 제시된 방법이 분석의 속도를 향상시킴을 보였다.

1. 서론

정적 분석(static analysis)이란 프로그램 실행 중의 관심있는 성질을 프로그램을 실행시키지 않고 미리 조사하는 것을 말한다. 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), 안전성 증명 등을 위하여 사용된다.

요약 해석(abstract interpretation)이란 래티스로 표현되는 요약된 공간에서 프로그램을 수행함으로써 프로그램의 성질을 파악하는 정적 분석 방법론을 말한다[1, 2, 3]. 이 방법론에서는 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 요약 해석은 함수 포인터(pointers)나 고차 함수(higher-order functions)와 같은 구조를 가지고 있는 프로그래밍 언어의 분석에 쉽게 적용될 수가 있으며, 요약의 정도를 조절함으로써 요구되는 분석 속도를 만족하도록 분석의 정밀

*본 연구는 과학기술부 창의적연구진흥사업의 지원을 받았음.

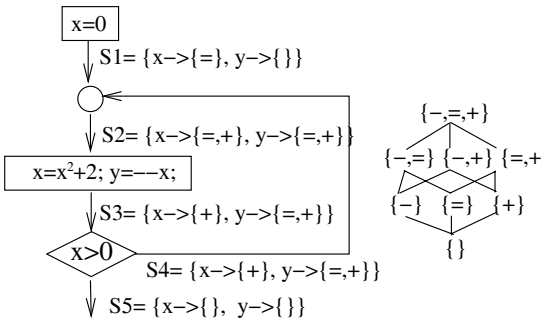

그림 1: 변수값의 부호 분석(sign analysis)

도를 조절할 수 있는 장점을 가진다.

요약 해석에 기반한 프로그램 분석에서 분석 시간의 대부분은 고정점 계산을 위하여 소모된다. 일반적으로 주어진 프로그램에 대한 정적 분석은 단조 증가 함수의 형태로 표현될 수 있으며, 이 함수의 고정점이 분석의 결과가 된다. 함수의 고정점을 구하기 위해서는, 결과값의 증가가 없을때까지 함수를 반복해서 적용하게 되며, 일반적으로 프로그램의 크기가 커지고 요구되는 분석의 정확도가 높아질수록 요구되는 계산의 양이 증가하게 된다. 또한 C나 ML과 같은 고차 언어의 요소를 가지고 있는 프로그래밍 언어의 경우에는 분석식 치환 방법(elimination method)[4]과 같은 효율적인 프로그램 흐름 분석 방법들이 적용될 수 없기 때문에, 고정점 계산의 속도 향상은 많은 연구자들의 관심을 받는 문제이다.

본 연구에서는 함수를 단순히 반복하여 적용하는 대신에 함수의 적용으로 인한 새로운 증가분만을 계산함으로써, 고정점 계산의 속도를 높이는 방법을 제시한다. 이를 통하여 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.

그림 1은 간단한 플로우차트 프로그램에 대한 분석의 예이다. 만약 우리가 각각의 에지(edge)에서 변수들의 값들이 가질 수 있는 부호를 모두 알고 싶다고 할 때, 그림의 오른편과 같은 요약된 공간에서 프로그램을 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 x값이 $\{-1,1,2\}$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 고정점에 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 y는 실제로 항상 양수인데 $\{=,+\}$로 분석된 것처럼, 어느 정도 부정확한 값을 생성하지만 가능한 모든 부호를 포함하는 안전한(sound) 분석 결과를 생성함을 볼 수 있다.

이러한 그림 1의 분석의 수행 과정을 보면, 반복된 분석 결과의 계산시에 중첩된(redundant) 계산이 수행되었음을 알 수 있다. 즉, S3의 x값의 계산에 있어서, 첫번째 계산에서 S2의 x값인 $\{=\}$으로 S3의 x값을 계산한 이후에, 두번째 S3의 x값의 계산에서 S2의 x값인 $\{=,+\}$를 가지고 계산을 수행할 때에 $=$ 원소에 대한 계산이 반복되어짐을 볼 수 있다. 본 연구에서는 S2의 x에 새로 첨가된 원소인 $+$만을 가지고 S3의 x의 새로운 값을 계산하도록 함으로써, 중복된 계산

을 최대한 없애고자 하였다.

증가분의 계산을 통한 프로그램 수행 속도의 향상은 이미 고전적인 접근법으로서, 일찌기 프로그램 최적화를 위한 기법인 인덕션(induction) 변수 대치-strength reduction-에 의하여 사용된 바 있다[5]. 이 최적화 기법에서는 반복문 내에서의 인덕션 변수를 검출해 내고, 이러한 인덕션 변수에 대한 계산을 이전의 변수값을 사용하는 효율적인 계산으로 대치함으로써 프로그램 수행의 속도를 높이고자 하였다. 이러한 연구의 연장선으로서 이러한 최적화를 집합 기반 언어의 집합 연산으로 확장하고자 하는 연구가 수행되었고[6], 증가분 계산에 의한 프로그램 최적화를 좀 더 일반화된 방법론으로 제시하고자 하는 연구가 수행되었다[7]. 그러나 이러한 연구 결과들은 최적화가 적용될 수 있는 자료 구조와 연산이 아직 제한적이기 때문에, 일반적인 프로그램 분석기에 자동으로 적용하는데는 어려움이 따른다.

본 연구와 좀 더 밀접한 기존 연구로서 고정점 계산을 증가분 계산에 기반하여 효율적으로 수행하는 방법으로는 추론 기반 데이터베이스(deductive database)의 결과를 효율적으로 얻어내기 위한 semi-naive method[8]가 제안되었다. 또한, Fecht등도 제약식의 해를 구하기 위한 고정점 계산 방법에서 이러한 접근법을 사용한 바가 있다[9]. 그러나 이러한 연구에서 제안한 방법은 분산 법칙을 만족하는 함수(distributive functions)의 고정점 계산만을 대상으로 하고 있다. 일반적인 프로그램 분석에서 얻어지는 함수들은 분산 법칙을 만족하지 않는 경우가 많으므로, 본 연구에서는 좀 더 일반적인 함수에 대하여 고정점 계산을 효율적으로 수행할 수 있는 방법을 제시하고자 하였다.

이어지는 절들에서는, 중첩된 계산을 피하는 고정점 계산 알고리즘의 원리와 증가값 계산을 주어진 함수에 대하여 자동으로 수행하기 위한 방법을 설명하고 실제 구현및 실험 결과를 제시한다.

2. 증가분 계산에 기반한 고정점 계산 알고리즘

요약해석에 기반한 주어진 프로그램의 분석은 우리가 관심을 갖고 있는 프로그램의 각각의 성질들 사이의 연립방정식으로 표현될 수 있다[10]. 예를 들어 그림 1에서 주어진 프로그램 분석은 다음과 같은 연립방적식으로 나타내어진다.

$$
\begin{aligned}
S_1 &= \{x \to \{=\},\ y \to \{\}\} \\
S_2 &= S_1 \sqcup S_4 \\
S_3 &= \{x \to \{s \hat{2} \hat{+} 2 \mid s \in S_2(x)\}, \\
&\qquad y \to \{\hat{-}(s \hat{2} \hat{+} 2) \mid s \in S_2(x)\} \\
S_4 &= \text{if } (S_3(x)) \hat{>} 0 \text{ then } S_3 \text{ else } \bot \\
S_5 &= \text{if } (S_3(x)) \hat{\leq} 0 \text{ then } S_3 \text{ else } \bot
\end{aligned}
$$

이 때, 분석의 결과는 위 연립방정식의 해가 되며, 연립방정식의 해는 위에서 주어진 $S_i = e_i$에 대하여 $F_i(S1, S2, S3, S4, S5) = e_i$로 F_i를 정의할 때$(1 \leq i \leq 5)$, 함수 $F = (F_1, F_2, F_3, F_4, F_5)$ 의 최소 고정점을 계산함으로써 얻을 수 있다.

함수 F가 단조 증가 함수이므로 함수 F의 최소 고정점은 다음과 같은 간단한 알고리즘에 의하여 구해진다$(\bot = (\{\}, \{\}, \{\}, \{\}, \{\}))$.

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

그런데 이러한 고정점 계산에는 함수 F를 계

속해서 증가하는 결과에 적용하기 때문에 이전에 계산되었던 값에 대한 중첩된 계산이 이루지게 된다. 즉, 앞에서 언급한 그림 1의 예를 보면, $S3$의 x 값의 계산을 위해서는 $S2$의 x값이 사용되는데, $S2$의 x값이 $\{=\}$일때와 $\{=,+\}$일때 $S3$의 값을 구하게 된다. 이때 x값 $=$에 대한 계산이 중첩되어 이루어짐을 알 수 있다. 이러한 중첩된 계산을 없애고 두번째 $S3$의 x값 계산시에는 새로이 포함된 $+$에 대한 계산만을 수행하여 이전값과 합쳐줌으로써 전체적인 계산의 양을 줄일 수 있다.

만약 주어진 함수 F에 대하여 다음과 같은 성질을 만족하는 효율적인 함수 F^δ를 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^δ는 v^δ만큼의 입력값 증가에 의한 함수 F의 결과값의 증가분을 구하는 함수이다. 만약 F가 $\sqcup$연산에 대하여 분산 법칙을 만족한다면 $F^\delta(v, v^\delta)$는 $F(v^\delta)$가 된다. 이와 같이 우리가 큰 v값과 상대적으로 작은 v^δ 값에 대하여 $F^\delta(v, v^\delta)$를 $F(v \sqcup v^\delta)$보다 훨씬 효율적으로 계산할 수 있다면, 그만큼 많은 중첩된 계산을 피하는 것이 된다. 최악의 경우, $F^\delta(v, v^\delta)$는 $F(v \sqcup v^\delta)$로 주어질 수 있으며, 이 경우에는 중첩된 계산을 없애지 못하는 것이 된다.

증가분을 계산하는 함수 F^δ를 사용한 고정점 알고리즘은 다음과 같다.

$$\begin{array}{l}
\mathtt{pv} = \bot \\
\mathtt{v} = \mathtt{dv} = F(\bot) \\
\mathtt{while}\ (\mathtt{v} \neq \mathtt{pv})\ \{ \\
\quad \mathtt{dv} = F^\delta(\mathtt{pv}, \mathtt{dv}) \\
\quad \mathtt{pv} = \mathtt{v} \\
\quad \mathtt{v} = \mathtt{pv} \sqcup \mathtt{dv} \\
\}
\end{array}$$

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 중첩된 계산을 피할 수 있게 된다.

일반적으로 프로그램의 분석에서 얻어지는 함수 F는 인자의 조인 연산에 대하여 분산 법칙을 만족하지 않기 때문에, 위의 알고리즘을 효과적으로 사용하기 위해서는 주어진 일반적인 F에 대하여 효율적인 F^δ 함수를 찾아내야 한다. 그러나, F^δ를 자동으로 생성해 내는 것은 쉽지 않으므로, 본 연구에서는 함수 몸체를 이루는 요약식(abstract expression)에 대한 계산 규칙(evaluation rules)과 증가분 계산 규칙(differential evaluation rules)을 사용하는 간접적인 방법을 사용하였다. 이어지는 절에서는 이러한 계산 규칙들과 이를 이용한 F^δ 함수값의 계산 방법에 대하여 설명한다.

3. 증가분 계산 함수 F^δ의 생성

주어진 프로그램의 정적 분석을 나타내는 함수의 계산을 정의하기 위하여, 먼저 함수의 몸체를 이루는 요약식의 형태를 정의한다.

그림 2는 요약식의 형태를 나타낸다. c는 상수이며, x는 변수, (e_1, e_2)는 튜플(tuple), $e.i$는

튜플 원소의 선택, $e_1 \sqcup e_2$는 조인 연산, $\Box e$는 e의 계산 결과로 얻어지는 집합의 원소들의 조인, $e_1[e_2/e_3]$는 집합의 원소를 래티스의 원소로 보내주는 함수 래티스의 원소 e_1의 e_3에 대한 값을 e_2로 지정하는 것을 나타내며, $\texttt{ap}\ e_1\ e_2$는 함수 래티스의 적용을 나타낸다. 그외에도 조건문, 지역변수 선언과 주어진 함수를 집합의 각각의 원소에 적용시키는 map 연산을 포함하고 있다. 식의 형태는 일차 함수형 언어(first-order functional languages)와 유사한 형태를 가지나, 값의 범위가 함수 래티스를 포함하기 때문에 고차 언어에 대한 분석을 나타낼 수가 있다.

요약식에 대한 계산 규칙은 변수들의 값을 저장한 환경과 요약식을 입력으로 받아 요약식의 값을 계산한다. $Eval$ 함수는 요약식의 각각의 형태에 대하여 귀납적으로 정의될 수 있는데 예를 들어 상수, 변수, 튜플식과 지역 변수 선언의 계산 방법은 다음과 같이 정의될 수 있다.

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

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

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

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

$$\begin{array}{rcl} e\in Exp & ::= & c \mid x \mid (e_1,e_2) \mid e.i \mid e_1\sqcup e_2\\ & & \mid \Box e \mid \texttt{map}\ f\ e\\ & & \mid \texttt{if}\ (e_0\le e_1)\ e_2\ e_3 \mid f\ e\\ & & \mid \texttt{let}\ x=e_1\ \texttt{in}\ e_2\ \texttt{end}\\ & & \mid e_1[e_2/e_3] \mid \texttt{ap}\ e_1\ e_2 \end{array}$$

그림 2: 요약식의 형태

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

이와 같이 주어진 계산 규칙에 대하여, 변수 환경의 증가에 따른 표현식의 값의 증가분은 증가분 계산 규칙 $Eval^{\Delta}(E,E^{\delta},e)$로 정의할 수 있다. 여기에서 E^{δ}는 변수값들의 증가분을 나타내는 변수 환경으로서, 증가분 계산 규칙 $Eval^{\Delta}$는 이전 변수 환경 E로부터 E^{δ}에서 저장된 값만큼 변수 환경이 증가 했을 경우 요약식 값의 증가분을 계산한다. 예를 들어, 상수, 변수, 튜플식, 지역 변수 선언식에 대한 증가분 계산 규칙은 다음과 같다.

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

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

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

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

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

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.

이와 같이 $Eval$과 $Eval^{\Delta}$를 정의할 경우 다음과 같은 성질을 증명할 수가 있다.

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.

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

위의 성질에 의해, $F(x) = e$로 정의된 함수 F에 대하여 $F(v)$와 $F^\delta(v, v^\delta)$를 각각 $Eval([x \rightarrow v], e)$와 $Eval^\Delta([x \rightarrow v], [x \rightarrow v^\delta], e)$로 계산할 경우 다음 성질을 만족하게 된다.

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

4. 워크리스트(worklist) 알고리즘

2절에서 제시된 증가분 기반 고정점 계산 알고리즘은 프로그램의 한 부분에 대한 분석값이 변화하였을 경우에 다른 모든 프로그램 부분들에 대한 분석 결과를 다시 계산하게 되므로 실제적인 구현에는 적합하지 못하다. 그래서, 새로 변화된 부분의 분석 결과에 영향을 받는 부분만을 다시 계산하는 고정점 계산 알고리즘이 제안되어 왔다[11, 12, 13].

본 연구에서도, 제시된 증가분 계산에 기반한 고정점 계산 방법을 적용하여 실용적인 고정점 계산 알고리즘을 설계하였다(그림 3). 제시된 알고리즘은 각각의 프로그램 부분들에 대하여 분석 결과의 증가분을 계산하고, 이로 인하여 해당 부분에 대한 분석값이 증가하였을 경우에, 이 값을 사용하는 프로그램 부분들을 worklist에 추가하여 다시 계산하게 된다.

5. 구현 및 실험

증가분의 계산에 기반한 고정점 계산 알고리즘의 성능을 실험하기 위하여 상수 분석 및 이명 분석을 동시에 수행하는 프로그램 분석을 요약해

```
/* V[i] : 계산 결과                    */
/* X[i] : 이전 계산 결과               */
/* dX[i] : 증가분 계산 결과            */
/* use[i] : X[i]값 사용 부분식들       */
/* worklist : 계산될 부분식들          */

for (i = 1 to n){
   V[i] ← Eval(X, e_i)
      /* use[] is updated.) */
   dX[i] ← V[i]
}
while (worklist ≠ φ) {
   i ← worklist
   dX[i] ← Eval^Δ(X, dX, e_i)
   X[i] ← V[i]
   if (dX[i] ⊔ V[i] ≠ V[i]){
      V[i] ← V[i] ⊔ dX[i]
      worklist ← append(worklist,
                        , use[i] − worklist)
   }
   else
      dX[i] ← ⊥
}
```

그림 3: 증가분 기반 워크리스트 알고리즘

석에 기반한 프로그램 분석도구인 Z1[14]을 사용하여 구현하였다. 그 실험 결과는 표 1과 같다. 괄호안의 숫자는 각각 프로그램 내의 프로시져의 갯수와 표현식의 갯수를 나타낸다. "단순 적용"은 기존의 방법을 사용한 경우, "증가분"은 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 증가분을 이전의 결과와 합쳐주는 계산으로 인한 부담(overhead)이 증가분만을 계산함으로써 얻어지는 계산양의 감소보다 더 컸기 때문으로 판단된다.

6. 결론

본 연구에서는, 요약해석에 기반한 프로그램 분석의 속도를 높이기 위하여, 증가분에 기반한 고정점 계산 방법을 제시하였다. 제시된 방법은 반복된 함수의 적용시에 이전 결과에 그대로 함수를 적용하는 것이 아니라, 인자값의 증가를 고려하여 결과의 증가분만을 계산하도록 함으로써 고정점 계산에 소요되는 계산의 양을 줄이게 된다. 그리고 제시한 방법을 실제적인 프로그램의 상수-이명 분석에 적용해 봄으로써, 프로그램 분석의 속도를 높일 수 있음을 보였다.

향후 연구는 다음과 같다. 우선 속도의 저하를 보인 프로그램들을 분석하여, 제시된 알고리즘의 부담을 정확히 분석해야 하며, 이를 극복할 수 있는 방법을 찾아내는 것이 필요하다. 만약 추가적인 부담을 없앨 수 없다면, 주어진 프로그램에 대하여 새로운 알고리즘의 적용 여부를 판단할 수 있는 방법을 제시해야 한다. 또한 상수-이명 분석 뿐 아니라 다른 다양한 분석에 대한 실험도 필요할 것으로 판단된다.

프로그램	증가분	단순 적용
simplex (44^1,8739^2)	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

1 : 프로그램의 함수 갯수
2 : 프로그램의 표현식 갯수

표 1: 상수-이명 분석 소요 시간 (단위:초)

참고문헌

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

[5] A. C. Fong and J. D. Ullman. Induction variables in very high-level languages. In *6th ACM Symposium on Principles of Programming Languages*, pages 104–112, 1976.

[6] Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, Vol.4, No.3, pages 402–454, 1982.

[7] Yanhong A. Liu. Efficiency by incrementalization : an introduction. *Higher-Order and Symbolic Computation*, Vol.13, No.4, pages 289–313, 2000.

[8] Francois Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *ACM SIGMOD Conference on Management of Data*, 1986, pages 16-52, 1986.

[9] 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.

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

[11] 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.

[12] 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.

[13] Christian Fecht and Helmut Seidl. A faster solver for general systems of equations, *Science of computer Programming*, Vol.35, No. 2, pages 137–161, 1999.

[14] Kwangkeun Yi. *Automatic Generation and Management of Program Analyses*. Ph.D. Thesis, Report UIUCDCS-R-93-1828. *Journal of Programming Languages*, Vol.3, No.1, pages 31–68, 1995.

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.