할 일들의 순서 선택이 자유로운 증가분 기반 고정점 계산 알고리즘
(A Differential Fixpoint Evaluation Algorithm for Arbitrary Worklist Scheduling)

안 준 선 †
(Joonseon Ahn)

요 약 본 연구에서는 증가분 기반 계산을 사용한 고정점 계산 방법을 제시하고 이에 기반한 새로운 워크리스트 알고리즘을 제시한다. 제시된 방법은 기존의 증가분 기반 계산과 달리 배분 법칙을 만족하지 않는 계산 시스템에도 효과적으로 적용될 수 있으며 증가분 기반 계산으로 인한 제약 조건을 만족하면서도 다양한 워크리스트 스케쥴링 방법을 사용할 수 있는 장점을 가지고 있다. 본 연구의 결과를 프로그램 정적 분석 방법인 요약 해석 방법에 적용하였으며, 이를 사용하여 상수 및 이명 분석과 메모리 생존 분석을 구현하였다. 제시된 실험 결과는 본 연구의 방법이 계산을 실제적으로 절약할 수 있음과, 적절한 워크리스트 스케쥴링 방법의 사용이 증가분 기반 계산에서도 중요함을 보여준다.

키워드 : 고정점 계산, 워크리스트 알고리즘, 증가분 기반 계산, 정적 분석, 요약 해석

Abstract We devise a differential fixpoint computation method and develop a new worklist algorithm based on it. Compared with other differential methods, our method can deal with non-distributive systems and adopt any worklist scheduling policy satisfying restrictions imposed by differential evaluation. As a practical application, we present an abstract interpretation framework and implement constant and alias analysis and memory lifetime analysis based on it. Our experiment shows that our method can save computation and worklist scheduling is also important in differential fixpoint evaluations.

Key words : fixpoint computation, worklist algorithm, differential evaluation, static analysis, abstract interpretation

1. 서 론

자료 흐름 분석(data flow analysis)이나 요약 해석(abstract interpretation), 집합 기반 분석(set-based analysis) 등 대부분의 프로그램 분석 방법론에서, 주어진 프로그램의 분석은 각각의 프로그램 부분들에 대한 분석 결과 사이의 관계를 기술하는 연립 방정식으로 표현할 수 있다[1]. 이 때 연립 방정식의 해는 프로그램의 안전한(sound) 분석 결과가 되며, 이러한 해는 연립 방정식으로부터 얻어지는 단조 증가 함수(monotonic increasing function)의 고정점 계산으로 얻을 수 있다.

프로그램 분석에서 분석 시간의 대부분은 고정점(fixpoints) 계산을 위하여 소모된다. 함수의 고정점을 구하기 위해서는 해당 도메인의 가장 작은 값(bottom)에서 시작하여 결과 값의 증가가 없을 때까지 함수를 반복해서 적용하게 되는데, 일반적으로 프로그램의 크기가 커지고 요구되는 분석의 정확도가 높아질수록 요구되는 계산의 양이 증가한다.

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, 전체 계산을 줄일 수가 있고, 결과적으로 분석의 속도를 높일 수가 있다. 제시된 방법은 함수가 배분 법칙(distributive law)을 만족하지 않는 경우에도 적용할 수 있는데, 이러한 경우는 실제적인 정적 분석에서 자주 발생한다. 또한 본 연구에서는 증가분 기반 계산을 사용한 워크리스트 알고

· 본 논문은 산업자원부 한국산업기술평가원 지정 "한국항공대학교 인터넷정보검색연구센터"의 연구비 지원으로 수행되었음
† 종신회원 : 한국항공대학교 항공전자 및 정보통신공학부 교수
jsahn@mail.hangkong.ac.kr
논문접수 : 2004년 6월 4일
심사완료 : 2005년 6월 2일

리즘을 제시하였다. 주어진 워크리스트 알고리즘은 다양한 스케쥴링 방법을 적용할 수 있다는 장점이 있는데, 적절한 워크리스트 스케쥴링의 중요성은 이미 관련된 연구에서 입증된 바 있다[2,3].

증가분의 기반 계산 방법은 큰 값을 사용한 계산이 작은 값을 사용한 계산보다 자원을 더 소모하는 경우에 흔히 사용된다. 그 첫 시도로는 프로그램 최적화를 위한 기법인 인덕션(induction) 변수 대치(strength reduction)를 들 수 있는데[4,5], 이 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 연장선으로 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 집합이나 리스트와 같은 자료 구조에 대한 반복적인 재귀 연산이나 줄이기 연산(reduction) 형태에 제한적으로 사용할 수 있기 때문에, 일반적인 프로그램 분석기에 자동으로 적용하기는 어렵다.

본 연구와 좀 더 밀접한 기존 연구로서 고정점 계산을 증가분 계산에 기반하여 효율적으로 수행하는 방법으로는 추론 데이타베이스(deductive database)의 결과를 효율적으로 얻어내기 위한 방법이 제안되었다[8]. 또한, Fecht 등은 집합 기반 분석(set-based analysis) 등에서 생성되는 제약식의 해를 구하기 위한 고정점 계산 방법에서 이러한 접근법을 사용한 바가 있다[9,10]. 그러나, 기존의 연구에서 제안한 방법은 배분 법칙을 만족하는 함수의 고정점 계산만을 대상으로 하고 있다.

본 논문의 구조는 다음과 같다. 2장에서는 본 논문의 대상이 되는 연립 방정식의 형태를 정의하고 증가분 기반 계산의 아이디어를 설명한다. 3장에서는 새로운 워크리스트 알고리즘을 제시한다. 4장에서는 본 연구의 한 응용으로서 증가분 기반 요약 해석 방법을 제시하고 5장에서 이에 기반한 상수/이명 분석을 기술한다. 6장에서는 관련된 구현과 실험 결과를 설명하고 7장에서 결론을 맺는다.

2. 문제 정의 및 기본 아이디어

본 연구에서 대상으로 하는 문제는 다음 조건을 만족하는 연립 방정식 $x_1 = e_1$, ..., $x_n = e_n$의 해를 구하는 것이다.

1) 변수 $x_i \in Var$는 완전 라티스 도메인(complete lattice domain) D의 원소를 값으로 갖는다.
2) $e_i \in Exp$는 변수들을 포함하는 식으로서 계산 규칙 $Eval: Env \times Exp \rightarrow D$를 갖는데, 계산 규칙은 주어진 계산식과 변수 환경 $Env: Var \rightarrow D$에 대하여 D의 원소를 유한 시간 내에 결과로 계산한다.
3) e_i의 계산은 환경에 대하여 단조 증가 한다. 즉 $\forall E_1, E_2 \in Env$에 대하여 $E_1 \sqsubseteq E_2$일 경우에 $Eval(E_1, e_i) \sqsubseteq Eval(E_2, e_i)$를 항상 만족하여야 한다. 여기서 $E_1 \sqsubseteq E_2$란 모든 변수 $x \in Domain(E_1)$에 대하여 $E_1(x) \sqsubseteq E_2(x)$가 항상 성립함을 말한다.

위 연립 방정식의 안전한 해는 함수 $F(\bar{x}) = (e_1, e_2, ..., e_n)$의 최소 고정점을 구함으로써 얻을 수 있는데 그 알고리즘은 다음과 같다($\bar{x} = (x_1, ..., x_n)$).

```
v̄ = ⊥̄      /* v̄ = (v1,...,vn), ⊥̄ = (⊥,...,⊥) */
do {
   v̄ ← F(v̄);
} until (v̄ is not increasing)
```

이 알고리즘에서 $F(\bar{v})$의 값은 변수 환경 $E = \{x_1 \rightarrow v_1, ..., x_n \rightarrow v_n\}$에 대하여 $(Eval(E, e_1), ..., Eval(E, e_n))$으로 구할 수 있다. 그런데 이러한 고정점 계산에서는 함수 F를 계속해서 증가하는 결과에 적용하기 때문에 이전에 계산되었던 값에 대한 중첩된 계산이 이루지게 된다. 일반적으로 집합 등의 라티스 원소에 대한 계산은 값이 증가하면서 더 많은 계산을 요구하므로 계산이 반복되면서 함수 적용에 더 많은 시간이 소요되게 된다. 그런데, 만약 주어진 함수 F에 대하여 다음과 같은 성질을 만족하는 효율적인 함수 F^{δ}를 찾아낼 수 있다면 고정점 계산에서 반복에 의한 중첩된 계산을 줄일 수 있다.

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

F^{δ}는 주어진 입력값의 증가에 따른 함수 F의 결과값의 증가분을 구하는 함수이다. 만약 F함수가 큰 값에 대하여 더 많은 계산을 필요로 하고, F^{δ}가 작은 v^{δ}에 대하여 효율적으로 증가분을 계산할 수 있다면, 위와 같은 증가분 계산을 통하여 $F(v \sqcup v^{\delta})$의 값을 효율적으로 계산할 수 있다. 이러한 F^{δ}가 주어진다면 우리는 다음과 같은 방법으로 반복된 계산을 피하면서 F의 고정점을 계산할 수 있다.

```
p̄v̄ ← ⊥̄
v̄ ← v̄^δ ← F(⊥̄)
while(v̄ ≠ p̄v̄) {
    v̄^δ ← F^δ(p̄v̄, v̄^δ)
    p̄v̄ ← v̄
    v̄ ← p̄v̄ ⊓ v̄^δ
}
```

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. 그러나 위의 알고리즘은 새로운 결과를 얻기 위하여 이전 결과와 새로운 증가분의 조인(join)연산을 수행해야 하는 부담(overhead)를 가지게 된다. 만약 증가분 계산으로 절약되는 계산량이 추가적인 조인 연산의 계산량보다 많다면 위의 증가분 계산 방법은 고정점 계산의 속도를 높일 수 있을 것으로 판단된다.

위의 증가분 기반 고정점 계산을 위해서는 위의 연립 방정식이 다음과 같은 추가적인 조건을 만족하여야 한다.

2') $e_i \in \mathrm{Exp}$는 다음을 만족하는 증가분 계산 규칙 $Eval^{\Delta} : Env \times Env \times \mathrm{Exp} \rightarrow D$를 갖는다.

$$\forall e \in \mathrm{Exp},\ Eval(E \sqcup E^{\delta}, e) = Eval(E, e) \sqcup Eval^{\Delta}(E, E^{\delta}, e)$$
$$where\ (E \sqcup E^{\delta})(x) = E(x) \sqcup E^{\delta}(x).$$

위 조건이 만족되면, $F(\bar{x}) = e$로 정의된 함수 F에 대하여 $F^{\delta}(\bar{v}, \bar{v}^{\delta})$를 다음과 같이 안전하게 계산할 수 있다.

$$(\bar{x} = (x_1, ..., x_n),\ \bar{v} = (v_1, ..., v_n),\ \bar{v}^{\delta} = (v_1^{\delta}, ..., v_n^{\delta}))$$

$$F^{\delta}(\bar{v}, \bar{v}^{\delta}) = Eval^{\Delta}(x_1 \rightarrow v_1, ..., x_n \rightarrow v_n, x_1 \rightarrow v_1^{\delta}, ..., x_n \rightarrow v_n^{\delta}, e)$$

3. 증가분 계산에 기반한 워크리스트 알고리즘

2장의 고정점 계산 알고리즘은 하나의 변수값이라도 증가할 경우에 모든 e_i들의 값을 다시 계산하므로 실용적이지 못하다. 그러므로 실제적인 구현에서는 증가한 변수값의 영향을 받는 식들만을 다시 계산하는 워크리스트(worklist) 알고리즘이 사용된다.

그림 1은 일반적인 워크리스트 알고리즘으로서 배열에 저장된 변수 환경 X에 대하여 worklist에 저장된 식 e_i의 값을 구하여 $X[i]$에 저장하는 작업을 반복한다. *worklist*에는 계산되어야 하는 식들의 인덱스를 저장하며, 식 e_i의 계산에 의하여 변수 x_i의 값이 증가할 경우 x_i의 값을 사용하는 식들의 인덱스가 *worklist*에 추가되어 재계산 된다. 이러한 방법을 통하여 재계산을 통하여 증가할 가능성이 있는 e_i들만을 계산하면서 고정점을 얻을 수 있다.

그림 1의 워크리스트 알고리즘의 표현식 값 계산을 증가분 계산에 기반한 방법으로 대치함으로써 그림 2의 증가분 계산에 기반 한 워크리스트 알고리즘을 얻을 수 있다. 이 알고리즘은 그림 1의 워크리스트 알고리즘에 각 식들의 증가분 및 이전 값의 초기화를 위한 부분을 추가하고, $X[i] \leftarrow Eval(X, e_i)$를 $dX[i] \leftarrow Eval^{\Delta}(X, dX, e_i)$; $V[i] = X[i] \sqcup dX[i]$로 대치함으로써 얻을 수 있는데, 여기에서 $V[i]$, $X[i]$, $dX[i]$는 각각 식 e_i의 가장 최근 계산된 값, 이전 값, 가장 최근의 증가분을 저장한다.

그러나 이러한 증가분 기반 워크리스트 알고리즘은 그림 1의 워크리스트 알고리즘과는 달리 적절한 증가분의 사용을 위하여 워크리스트의 스케줄링과 관련하여 추가적인 조건을 고려하여야 한다. 예를 들어 식 e_3의 값이 증가하여 변수 x_3를 사용하는 식 e_1과 e_2가 워크리스트에 추가되었다고 가정하자. 이에 대하여 다음과 같은 경우가 발생할 수 있다.

1) e_1과 e_2가 e_3의 재계산 전에 모두 워크리스트에서 추출되어 계산될 경우에 $X[3]$와 $dX[3]$는 증가분의 중복 사용을 피하기 위하여 각각 $V[3]$, $\bot$으로 갱신된다.
2) e_3가 e_1과 e_2의 계산 전에 다시 계산되어 dv'만큼 증가한다면 $dX[3]$는 이를 반영하여 각각 $dX[3] \sqcup dv'$으로 갱신된다.
3) 만약 e_1만 워크리스트로부터 추출되어 계산 된 후에

```
/*  X[i] : 변수 x_i 의 값을 저장 (⊥ 으로 초기화 됨)         */
/*  use[i] : x_i 의 값을 사용하는 표현식들의 집합           */
/*  worklist : 계산될 식들의 집합                           */
worklist ← {1, 2, ..., n}                     /* 모든 표현식들의 집합으로 초기화 */
while (worklist ≠ φ) {
  i ← Extract(worklist)              /* 계산할 식을 선택하여 worklist에서 꺼냄 */
  x ← Eval(X, e_i)      /* Eval(X, e_i) 가 X[j] 의 값을 사용하면 i 를 use[j] 에 추가 */
  if (x ≠ X[i]) {                                      /* X[i] 의 값이 증가할 경우 */
    X[i] ← x
    worklist ← worklist ∪ use[i]
  }
}
```

그림 1 일반적인 고정점 계산 알고리즘

```
/*  V[i] : 변수 x_i 의 현재 값을 저장 (⊥ 으로 초기화 됨)                 */
/*  X[i] : 변수 x_i 의 가장 최근 이전값을 저장 (⊥ 으로 초기화 됨)          */
/*  dX[i] : 변수 x_i 의 까장 최근 증가분을 저장 (⊥ 으로 초기화 됨)        */
/*  use[i] : x_i 의 값을 사용하는 표현식들의 집합                          */
/*  worklist : 계산될 식들의 집합                                          */
worklist ← {1,2,...,n}                        /* 모든 표현식들의 집합으로 초기화 */
for (i = 1 to n){                                          /* V[ ]와 dX[ ]의 초기화 */
  V[i] ← Eval(X, e_i)      /* Eval(X,e_i) 가 X[j] 의 값을 사용하면 i 를 use[j] 에 추가 */
  dX[i] ← V[i]
}
while (worklist ≠ φ) {
  i ← Extract(worklist)                    /* 계산할 식을 선택하여 worklist에서 꺼냄 */
  dV ← Eval^Δ(X, dX, e_i)/* Eval^Δ(X,dX,e_i) 가 X[j] 를 사용하면 i 를 use[j] 에 추가 */
  New_V ← X[i] ⊔ dV
  if (New_V ≠ V[i]) {                                      /* V[i] 의 값이 증가할 경우 */
    X[i] ← V[i]
    V[i] ← New_V
    dX[i] ← dV
    worklist ← worklist ∪ use[i]
  }
}
```

그림 2 증가분 계산 규칙을 사용한 워크리스트 알고리즘

e_3가 다시 계산되어 dv'만큼 증가한다면 e_1과 e_2의 계산을 위한 e_3의 증가분은 각각 dv'과 $dX[3] \sqcup dv'$이 된다.

증가분에 기반 하지 않은 고정점 계산의 경우 가장 최근에 계산된 변수값만 가지고 있으면 되므로 이러한 문제가 발생하지 않는데 비하여, 증가분 기반 계산에서는 계산을 위한 각각의 변수의 증가분이 해당 변수를 사용하는 식에 따라 달라질 수 있다. 이러한 문제에 대하여 기존의 해결 방법은 다음과 같다[9-11].

1) 워크리스트로 큐(Queue)를 사용한다. 큐를 사용하면, 워크리스트에 중복된 원소를 허용하지 않을 경우, 위의 예에서 1)의 경우만이 발생하므로 증가분을 식별로 따로 관리할 필요가 없다. 그러나 워크리스트 알고리즘에서 워크리스트 내의 식들의 계산 순서 스케줄링은 고정점 계산의 속도에 큰 영향을 미치는데, 큐를 사용하게 되면 스케줄링 방법이 고정된다.
2) 모든 식에 대하여 사용하는 변수들의 증가분을 따로 저장한다. 즉 위의 예에서 3)의 경우, e_1과 e_2의 계산을 위한 e_3의 증가분을 각각 dv'과 $dX[3] \sqcup dv'$로 따로 저장하게 된다. 이와 비슷한 접근법은 Fecht와 Seidl이 배분 법칙이 성립하는 방정식의 계산에서 사용하였다[9,10]. 이 방법은 워크리스트 스케쥴링이 자유로운 장점은 있으나, 추가적인 메모리 부담이 요구되고 어떤 식의 값이 증가하였을 경우에 해당 식의 값을 사용하는 모든 식들을 위한 증가분 값을 각각 조인 연산을 사용하여 일일이 계산하여야 하는 단점이 존재한다.

본 연구에서는 위와 같은 단점을 해결하기 위하여 기존의 워크리스트 알고리즘과 다른 접근법을 사용하였다. 기존의 워크리스트 알고리즘은 새로 계산되어야 할 식들을 워크리스트에 추가하는데 반해, 제안된 알고리즘은 계산되어 값이 증가된 식을 워크리스트에 추가한다. 그리고 워크리스트에서 어떤 식이 추출되면 해당 식의 값을 사용하는 식들을 모두 계산하게 된다. 이렇게 함으로써 어떤 식의 값을 사용하는 모든 식들에 대하여 따로따로 증가분을 저장하지 않으면서도, 워크리스트 내의 계산들의 스케줄링을 자유롭게 수행할 수 있다.

그림 3은 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 증가하면 $use[i]$에 포함된 식들의 값은 새로 계산되어야 한다.

$worklist$는 계산에 의하여 값이 증가한 식들을 저장한다. 기존의 알고리즘들은 e_i의 값이 증가하면 $use[i]$에 있는 식들이 저장되었으나, 본 알고리즘에서는 e_i가 증가하면 i를 저장하고 후에 워크리스트에서 저장된 i가 선택되면 $use[i]$에 속한 식들이 모두 계산되게 된다. 이러한 방법을 통하여 계산 순서의 부분적인 제약을 유지하면서 워크리스트 내의 계산할 표현식의 선택 순서를 자유롭게 할 수 있다. 또한 모든 식에 대하여 최근의 계산 시점을 $\times tamp[i]$에 저장하여, 이미 최신의 값을 사용하여 계산된 경우에 중복된 계산을 피하도록 하였다. 이러한 계산 시점은 가장 바깥쪽 반복문의 반복 횟수로 큰 추가 부담 없이 구현될 수 있다.

```
/* V[i] : 변수 x_i 의 현재 값을 저장 (⊥ 으로 초기화 됨)                 */
/* X[i] : 변수 x_i 의 가장 최근 이전값을 저장 (⊥ 으로 초기화 됨)          */
/* dX[i] : 변수 x_i 의 가장 최근 증가분을 저장 (⊥ 으로 초기화 됨)         */
/* use[i] : x_i 의 값을 사용하는 표현식들의 집합                          */
/* worklist : 값이 증가한 식들의 집합                                    */
/* timestamp[i] : 표현식 i의 최근 계산 시점                              */
worklist ← {1,2,...,n}
eval_count ← 0
for (i = 1 to n){
  V[i] ← Eval(X, e_i)      /* Eval(X,e_i) 가 X[j] 의 값을 사용하면 i 를 use[j] 에 추가 */
  dX[i] ← V[i]
}
while (worklist ≠ φ) {
  i ← Extract(worklist)
  foreach (w ∈ use[i])
    if (timestamp[i] > timestamp[w]) {
      timestamp[w] ← eval_count ++
      dV ← Eval^Δ(X, dX, e_w)
              /* Eval^Δ(X, dX, e_i) 가 X[j] 를 사용하면 i 를 use[j] 에 추가 */
      New_V ← V[w] ⊔ dV
      if (New_V ≠ V[w]) {
        worklist ← worklist ∪ {w}
        dX[w] ← dX[w] ⊔ dV
        V[w] ← New_V
      }
    }
  X[i] ← V[i]
  dX[i] ← ⊥
}
```

그림 3 증가분 기반 워크리스트 알고리즘

4. 요약 해석을 위한 증가분 기반 고정점 계산

본 절에서는 증가분 기반 고정점 계산의 한 응용으로서 요약 해석에 기반 한 정적 분석을 제시한다. 요약 해석이란 실제 프로그램 수행 상황을 완전 라티스 도메인의 값으로 안전하게 근사(approximate)하고, 이 안에서 프로그램을 수행함으로써 프로그램 실행시의 성질을 분석하는 방법이다. 따라서 하나의 정적 분석은 대상 프로그래밍 언어에 대한 요약된 의미(semantics)로 표현되며, 프로그램의 분석은 요약된 의미에 대하여 프로그램을 실행하는 것이 된다[12,13].

입력 언어에 대한 요약된 의미는 일반적으로 각각의 언어 구조(language constructs)에 대하여 해당 언어 구조의 실행 전 상황에 대하여 실행 후의 상황을 재귀적으로 정의하는 방식으로 기술되는데, 일반적으로 묘사되는 상황은 해당 프로그램 구조 실행 전후의 메모리 상태와 분석과 관련된 정보를 안전(sound)하게 근사하는 형태로 표현된다. 분석할 프로그램이 주어지면 요약된 의미에 기반 하여 프로그램 내의 모든 부분들의 수행 상황간의 관계를 나타내는 연립 방정식을 생성해 낼 수 있으며, 이 연립 방정식의 해는 프로그램 분석의 안전한 결과가 된다.

본 절에서는 일반적인 프로그램 분석을 묘사할 수 있는 요약식의 형태를 정의하고, 해당 요약식에 대한 계산 규칙과 증가분 계산 규칙을 제시한다. 정의된 요약식으로 기술한 프로그램 의미와 주어진 프로그램으로부터 생성되는 연립 방정식의 식들은 요약식의 형태를 가지므로, 제시된 계산 규칙을 사용하여 계산할 수 있다. 정의된 요약식을 사용하여 기술한 정적 분석의 예는 5절에서 제시하기로 한다.

4.1 요약식(A Abstract Expression Language)

요약식은 그림 4의 형태를 가지는데, 각각 상수, 변수, 기본 연산, 조인 연산, 튜플식과 튜플식의 원소 추출, 집합값 생성, 집합의 각 원소에 대한 함수 적용, 함수값 생성 및 갱신, 함수 적용(function application), 조건문, 지역 변수의 사용을 나타낸다. 위와 같이 정의된 요약식을 사용하여 요약 해석에 기반한 정적 분석을 기술할 수 있으며, 위 표현식과 비슷한 형태로 Z1이 제안되어 실제적인 프로그램 정적 분석의 정의에 사용된 바 있다[14].

그림 6은 주어진 변수 환경 $Env : Vars \rightarrow D$에 대하여 위 표현식의 값을 계산하는 규칙 $Eval : Env \times \mathrm{Exp} \rightarrow D$를 정형적으로 정의하고 있다. 계산 규칙은 위 표현식의 각각의 형태에 대하여 재귀적으로 정의된다. 상수식 c는 해당하는 라티스 원소인 c로 계산되며, 변수의 값은 변

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

그림 4 요약식 문법

$S_\perp^\top$	최고 및 바닥 값이 추가된 집합 도메인(a flat domain)
$D_1 \times \cdots \times D_n$	곱 도메인(a product domain)
2^S	모든 부분집합 도메인(a powerset domain of a set S)
$S \rightarrow D$	함수 도메인(a function domain)

그림 5 요약식으로 표현 가능한 라티스 공간

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

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

$$\frac{Eval(E, e) = v \ \ (v \in (S_\perp^\top - \{\perp, \top\}))}{Eval(E, \{e\}) = \{v\}}$$

$$\frac{Eval(E, e') = \{v_1, \ldots, v_n\},\ Eval(E + \{x \rightarrow v_i\}, e) = \{v_1', \ldots, v_n'\}}{Eval(E, \texttt{mapjoin}\ \lambda x.e\ e') = v_1' \sqcup \ldots \sqcup v_n'}$$

$$\frac{Eval(E, e_i) = v_i \ \ (1 \leq i \leq n)}{Eval(E, (e_1, \ldots, e_n)) = (v_1, \ldots, v_n)} \qquad \frac{Eval(E, e) = (v_1, \ldots, v_n)}{Eval(E, e.i) = v_i \ \ (1 \leq i \leq n)}$$

$$\frac{Eval(E, e_i) = v_i \ \ (i = 1, 2,\ \ v_2 \in (S_\perp^\top - \{\perp, \top\}))}{Eval(E, \texttt{ap}\ e_1\ e_2) = @\ v_1\ v_2, \text{ where } \begin{array}{lcll} @\ \perp_{S \rightarrow D}\ v & = & \perp_D & \\ @\ v_0[v_1/v_2]\ v & = & v_1 & \text{if } v = v_2 \\ & = & @\ v_0\ v & \text{otherwise} \end{array}}$$

$$\frac{Eval(E, e_i) = v_i \ \ (i = 1, 2, 3,\ \ v_3 \in (S_\perp^\top - \{\perp, \top\}))}{Eval(E, e_1[e_2/e_3]) = v_1[v_2/v_3]}$$

$$\frac{Eval(E, e_i) = v_i \ \ (1 \leq i \leq 4)}{Eval(E, \texttt{if}\ (e_0 \sqsubset e_1)\ e_2\ e_3) \begin{array}{ll} = v_2 & \text{if } v_0 \sqcup v_1 = v_1 \\ = v_3 & \text{otherwise} \end{array}}$$

$$\frac{Eval(E, e) = v, \quad Eval(E + \{x \rightarrow v\}, e') = v'}{Eval(E, \texttt{let}\ x = e\ \texttt{in}\ e'\ \texttt{end}) = v', \text{ where } (E_1 + E_2)(x) \begin{array}{lll} = & E_2(x) & \text{if } x \in dom(E_2) \\ = & E_1(x) & \text{otherwise} \end{array}}$$

그림 6 요약식의 계산 규칙($Eval$: $Env \times \mathrm{Exp} \rightarrow D$)

수 환경으로부터 얻어진다. op는 연산자 op의 의미를 나타내며 오퍼랜드의 계산 결과에 해당 연산을 수행하며, $e_1 \sqcup e_2$는 e_1의 결과값과 e_2의 결과값을 조인한 결과를 생성한다. e는 요약식 e의 값을 원소로 가지는 집합값을 생성하며, mapjoin 식은 함수를 두 번째 인자로부터 생성되는 집합값의 각각의 원소에 적용한 후 그 결과를 모두 조인하는 연산을 수행한다. $(e_1,\ldots,e_n)$은 각각의 부분식의 결과로부터 튜플 도메인의 값을 생성하며, $e.i$는 e로부터 생성된 튜플값의 i번째 값을 결과로 계산한다. $e_1[e_3/e_2]$는 e_1으로부터 생성된 함수값에 대하여 e_2로부터 계산되는 값에 대해서는 e_3를 돌려주도록 갱신한 결과를 생성하며, ap문은 함수값 적용을 수행한다. if 문은 조건 부분을 계산한 결과에 따라 두 선택중 하나의 요약식을 계산하여 결과를 생성하며, let문은 지역 변수 바인딩을 나타낸다.

4.2 증가분 계산 규칙

주어진 계산 규칙 $Eval$에 대하여, 변수 환경의 증가에 따른 표현식의 값의 증가분을 계산하는 규칙 $Eval^{\Delta}$는 그림 7과 같이 정의된다. 여기에서 E^{δ}는 변수값들의 증가분을 나타내는 변수 환경으로서, 증가분 계산 규칙

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

$$\frac{Eval(E,e)=v, \quad Eval^{\Delta}(E,E^{\delta},e)=v'}{Eval^{\Delta}(E,E^{\delta},op\ e) \ = \ \begin{array}{ll} \mathbf{op}\ v' & \text{if op is distributive w.r.t. } \sqcup \\ \mathbf{op}\ (v \sqcup v') & \text{otherwise} \end{array}}$$

$$\frac{Eval^{\Delta}(E,E^{\delta},e_i)=v_i \ (i=1,2)}{Eval^{\Delta}(E,E^{\delta},e_1 \sqcup e_2)=v_1 \sqcup v_2} \qquad \frac{Eval^{\Delta}(E,E^{\delta},e)=(v_1,\ldots,v_n)}{Eval^{\Delta}(E,E^{\delta},e.i)=v_i \ (1 \le i \le n)}$$

$$\frac{Eval^{\Delta}(E,E^{\delta},e_i)=v_i \ (1 \le i \le n)}{Eval^{\Delta}(E,E^{\delta},(e_1,\ldots,e_n))=(v_1,\ldots,v_n)} \qquad Eval^{\Delta}(E,E^{\delta},\{e\})=\{\}$$

$$\frac{\begin{array}{l} Eval(E,e')=\{v_1,\ldots,v_n\}, \quad Eval^{\Delta}(E,E^{\delta},e')=\{v'_1,\ldots,v'_m\}, \\ Eval^{\Delta}(E+\{x \to v_i\},E^{\delta}+\{x \to v_i\},e)=v_i^{\delta} \ (1 \le i \le n), \\ Eval((E \sqcup E^{\delta})+\{x \to v'_i\},e)=v''_i \ (1 \le i \le m) \end{array}}{Eval^{\Delta}(E,E^{\delta},\mathtt{mapjoin}\ (\lambda x.e)\ e')=v_1^{\delta} \sqcup \ldots \sqcup v_n^{\delta} \sqcup v''_1 \sqcup \ldots \sqcup v''_n}$$

$$\frac{Eval^{\Delta}(E,E^{\delta},e_i)=v_i \ (i=1,2), \quad Eval(E,e_3)=v_3}{Eval^{\Delta}(E,E^{\delta},e_1[e_2/e_3])=v_1[v_2/v_3]}$$

$$\frac{Eval^{\Delta}(E,E^{\delta},e_1)=v_1, \quad Eval(E,e_2)=v_2}{Eval^{\Delta}(E,E^{\delta},\mathtt{ap}\ e_1\ e_2)=(@\ v_1\ v_2)}$$

$$\frac{Eval(E,e_i)=v_i, \ Eval^{\Delta}(E,E^{\delta},e_1)=v_i^{\delta}, \ (1 \le i \le 4)}{\begin{array}{l} Eval^{\Delta}(E,E^{\delta},\mathtt{if}\ (e_0 \sqsubset e_1)\ e_2\ e_3) \\ \quad = v_2^{\delta} \quad \text{if } (v_0 \sqcup v_1)=v_1 \text{ and } (v_0 \sqcup v_0^{\delta}) \sqcup (v_1 \sqcup v_1^{\delta}) = (v_1 \sqcup v_1^{\delta}) \\ \quad = v_3^{\delta} \quad \text{if } (v_0 \sqcup v_1) \ne v_1 \text{ and } (v_0 \sqcup v_0^{\delta}) \sqcup (v_1 \sqcup v_1^{\delta}) \ne (v_1 \sqcup v_1^{\delta}) \\ \quad = Eval((E \sqcup E^{\delta}),\mathtt{if}\ (e_0 \sqsubset e_1)\ e_2\ e_3) \quad \text{otherwise} \end{array}}$$

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

그림 7 요약식의 증가분 계산 규칙($Eval^{\Delta}: Env \times Env \times Exp \to D$)

$Eval^{\Delta}$는 이전 변수 환경 E로부터 E^{δ}에서 저장된 값만큼 변수 환경이 증가했을 경우 요약식 값의 증가분을 계산한다.

그림 7에서 정의한 증가분 계산 규칙은 다음과 같은 성질을 만족하며, 이는 정의된 규칙이 환경의 증가에 따른 안전한 증가분을 계산함을 나타낸다. 다음 정리에서 전체적 단조 증가(entirely monotonic)란 주어진 식의 모든 부분식이 단조 증가 성질을 갖는 것을 뜻한다.

정리 1. 변수 환경 $\forall E, E^{\delta} \in Env$와 전체적 단조 증가의 성질을 가진 식 $e \in E$에 대하여 $Eval(E,e)=v$이고 $Eval(E \sqcup E^{\delta}, e)=v'$이면 $v'=v \sqcup Eval^{\Delta}(E, E^{\delta}, e)$가 항상 성립한다.

위 정리는 표현식 e의 각각의 형태에 대한 구조적 귀납법(induction)으로 쉽게 증명할 수 있다.

5. 상수 및 이명 분석의 구현

본 장에서는 이전 장에서 제시한 요약 해석 기법에 기반 한 프로그램 분석의 한 예로서 상수 및 이명 분석을 제시한다. 상수 및 이명 분석은 다음과 같은 사항을 실행전에 알아내는 것을 목적으로 한다.

- 상수 분석 : 프로그램 내의 한 지점에서 어떤 변수의 값이 항상 같은 값을 가지고 있는가?
- 이명 분석 : 프로그램 내의 한 지점에서 어떤 변수들이 같은 메모리 위치를 나타내고 있는가?

이 분석을 위한 요약된 의미는 이미 발표되었으며 [14], 본 장에서는 정적 분석의 방정식적인 측면을 보이기 위하여 주어진 프로그램으로부터 분석 결과를 계산하기 위한 방정식을 생성하는 규칙을 제시한다. 또한 제시된 규칙으로부터 생성되는 방정식의 형태로부터 증가분 기반 계산 방법이 실제적으로 어떤 이득을 줄 수 있는지를 설명한다.

5.1 입력 언어

분석이 대상이 되는 입력 언어는 MIL(MIPRAC Interprocedural Language[15])을 사용하였다. MIL은 실용적인 프로그래밍 언어의 구현을 위한 중간 언어로서, ANSI-C, Fortran, Scheme 등의 고급 프로그래밍 언어

```
e ∈ Mil  ::=  const c | + e1 e2 | id | create e1 id | read e1 | write e1 e2
           |  procedure (x1 ... xn) e1 | call e1 e2 ... en
           |  begin e1 ... en | if e1 e2 e3
```

그림 8 입력 언어 : MIL

를 위한 전단부(front-ends)를 가지고 있다.

그림 8은 MIL의 문법이다. c는 정수 상수를 나타내며, +는 원시 연산을 나타낸다. id는 메모리 위치를 나타내며 create 문은 id를 위한 주어진 크기의 메모리를 할당한다. read와 write 구조는 메모리 위치의 값을 읽거나 쓰는데 사용되며, procedure 문은 함수 클로저(function closure)를 생성하고, call 문은 함수 적용(function application)을 나타낸다. begin 구조는 각각의 식들을 하나씩 순서대로 계산하며 마지막 식의 계산 결과가 전체 계산 결과가 된다. MIL 언어에서 반복문은 재귀 호출로 표현한다.

X_i^+	∈	*Prestate*	:=	*Memory*
X_i^-	∈	*Poststate*	:=	*Memory* × *Value*
		Memory	:=	*SetId* → *Value*
		Value	:=	*Loc* × *Clo* × *Int*
		Loc	:=	2^{SetId}
		Clo	:=	$2^{SetProc}$
		Int	:=	2^{SetInt}

그림 9 분석을 위한 라티스 공간

5.2 분석을 위한 도메인(domains)

분석을 위한 도메인은 그림 9와 같다. X_i^+와 X_i^-는 식 e_i의 실행 전 상태와 실행 후의 상태를 나타낸다. 또한 $X_i^-.1$은 실행 후의 메모리 상태를 나타내며 $X_i^-.2$는 실행 결과를 나타낸다. Memory 는 실제 메모리 상태를 나타내는 함수 도메인의 값으로서 메모리 셀을 나타내는 *SetId* 집합의 원소에 대하여 결과값을 나타내는 *Value* 값을 돌려준다. 각각의 메모리 셀은 포인터 값이나 함수 클로저, 정수값을 저장할 수 있기 때문에 *Value* 도메인은 *Loc, Clo, Int* 도메인의 튜플(tuple)로 표현되며 *Loc, Clo, Int*는 실제 프로그램 실행 시 가능한 값을 모두 표현하기 위해 각각 *SetId, SetProc, SetInt* 도메인의 파워셋(power set) 도메인으로 정의된다.

5.3 방정식 생성 규칙

그림 9의 도메인을 사용하여 MIL언어의 상수 및 이명 분석을 위한 요약된 의미를 정의할 수 있다. 본 절에서는 프로그램 각 부분의 실행 상태 간의 관계를 나타내는 방정식의 생성 규칙을 제시하며, 요약된 의미는 이 규칙으로부터 쉽게 유추할 수 있으므로 생략한다.

그림 10은 식을 생성하기 위한 규칙이다. 이 규칙은 주어진 프로그램의 모든 부분식들에 적용되어 각 부분식의 수행 상태들 간의 관계를 나타내는 식을 생성하며, 이렇게 생성된 방정식의 해를 구함으로써 정적 분석의 결과를 얻을 수 있다. 식을 이루는 모든 부분식들은 단조 증가 조건을 만족하며, 그림 4에서 제시한 요약식의 형태를 가진다.

const 식은 실행 후의 메모리 상태는 이전과 동일하며 수행 결과는 해당 상수 식에 대한 정수 값이 된다. id 역시 메모리의 상태는 이전과 동일하며 결과값은 해당 식별자의 메모리 위치를 포함한다.

$+ e_{i_1} e_{i_2}$는 연산자의 계산을 나타내며 먼저 e_{i_1}을 계산하고 e_{i_2}를 계산하므로 e_{i_1}의 실행 후의 메모리 상태가 e_{i_2}의 실행 전의 메모리 상태가 되며, +연산은 메모리 상태를 변화시키지 않으므로 e_{i_2} 실행 후의 메모리 상태가 전체 식의 실행 후의 메모리 상태가 된다. 연산자의 실행 결과는 가능한 모든 값을 포함하여야 하므로 e_{i_1}과 e_{i_2}의 계산 결과들의 모든 쌍에 대하여 해당 연산자를 적용한 결과를 모두 결과값에 포함시킨다.

create e_{i_1} *id*는 *id*를 위한 e_{i_1}크기의 메모리를 새로 할당하는 명령어로서, 메모리 상태는 이전과 같으며, 결과는 새로운 메모리 위치를 포함한다. 본 분석을 위한 요약된 의미에서는 메모리의 크기에 대한 정보를 제거하였다.

read e_{i_1}은 e_{i_1}이 가리키는 메모리 위치의 값을 읽어오는 연산이다. e_{i_1}은 실제 실행시 가능 한 모든 메모리 위치를 포함하므로 안전한 연산을 위하여 read 연산은 이러한 메모리 위치를 모두 읽어서 그 결과들을 조인하여 결과값을 생성한다.

write $e_{i_1} e_{i_2}$는 e_{i_1}식이 나타내는 메모리 위치를 e_{i_2}의 식의 값으로 갱신하는 연산으로서, 요약된 의미는 모든 가능한 경우를 포함하여야 하므로 e_{i_1}의 가능한 모든 메모리 위치에 대하여 갱신을 수행한 메모리 상태를 모두 조인(join) 함으로써 안전한 결과를 생성한다.

begin $e_{i_1} \dots e_{i_n}$은 각각의 부분식들을 순서대로 계산하므로 이전 식의 수행 후 메모리 상태가 다음 식의 수행 전 메모리 상태가 되며 전체 식의 수행 결과는 e_{i_n}의 수

$$e_i = \texttt{const}\ c \;\Rightarrow\; X_i^- = (X_i^+, (\bot_L, \bot_C, \{c\}))$$
$$e_i = id \;\Rightarrow\; X_i^- = (X_i^+, (\{Loc(id)\}, \bot_C, \bot_Z))$$
$$e_i = +\ e_{i_1}\ e_{i_2}$$
$$\Rightarrow\; X_{i_1}^+ = X_i^+;\;\; X_{i_2}^+ = X_{i_1}^-.1;$$
$$X_i^- = (X_{i_2}^-.1,$$
$$(\bot_L, \bot_C, \texttt{mapjoin}\ (\lambda x.(\texttt{mapjoin}\ (\lambda y.\{x+y\})\ X_{i_2}^-.2.3))\ X_{i_1}^-.2.3))$$
$$e_i = \texttt{create}\ e_{i_1}\ id \;\Rightarrow\; X_{i_1}^+ = X_i^+;\; X_i^- = (X_{i_1}^-.1, (\{NewLoc(id)\}, \bot_C, \bot_Z))$$
$$e_i = \texttt{read}\ e_{i_1}$$
$$\Rightarrow\; X_{i_1}^+ = X_i^+;\;\; X_i^- = (X_{i_1}^-.1,\ \texttt{mapjoin}\ (\lambda x.(\texttt{ap}\ (X_{i_1}^-.1)\ x))\ X_{i_1}^-.2.1)$$
$$e_i = \texttt{write}\ e_{i_1}\ e_{i_2}$$
$$\Rightarrow\; X_{i_1}^+ = X_i^+;\;\; X_{i_2}^+ = X_{i_1}^-.1;$$
$$X_i^- = (\texttt{mapjoin}\ (\lambda x.X_{i_2}^-.1[(\texttt{ap}\ X_{i_2}^-.1\ x)/x])\ X_{i_1}^-.2.1,\;\; X_{i_2}^-.2)$$
$$e_i = \texttt{procedure}\ (x_1\ \ldots\ x_n)\ e_{i_1}$$
$$\Rightarrow X_{i_1}^+ = \;\texttt{mapjoin}$$
$$(\lambda j.(X_{lastsub(j)}^-.1)[X_{sub(j,2)}^-.2/Loc(x_1)]\ldots[X_{sub(j,n+1)}^-.2/Loc(x_n)])$$
$$(\texttt{mapjoin}\ (\lambda j.\mathrm{if}\ (\{p\} \sqsubset X_{subexp(j,1)}^-.2.2)\ \{j\}\ \{\ \})\ Calls^n);$$
$$\text{where } Calls^n = \{j \mid e_j = \texttt{call}\ e_{j_0}\ldots e_{j_m} \text{ and } n \le m\}$$
$$X_i^- = (X_i^+.1, (\bot_L, \{ProcId(i)\}, \bot_Z))$$
$$e_i = \texttt{call}\ e_{i_0}\ e_{i_1}\ \ldots e_{i_n}\ (n \ge 0)$$
$$\Rightarrow\; X_{i_0}^+ = X_i^+;\;\; X_{i_j}^+ = X_{i_{j-1}}^-.1\ (1 \le j \le n);$$
$$X_i^- = \texttt{mapjoin}\ (\lambda x.\mathrm{if}\ (\{NumOfParam(x)\} \sqsubset \{1,\ldots,n\})\ X_{body(x)}^-\ \bot)$$
$$X_{i_0}^-.2.2$$
$$e_i = \texttt{begin}\ e_{i_1}\ldots e_{i_n} \;\Rightarrow\; X_{i_1}^+ = X_i^+;\; X_i^- = X_{i_n}^-;\; X_{i_j}^+ = X_{i_{j-1}}^-.1\ (2 \le j \le n)$$
$$e_i = \texttt{if}\ e_{i_1}\ e_{i_2}\ e_{i_3}$$
$$\Rightarrow\; X_{i_1}^+ = X_i^+;\;\; X_{i_2}^+ = X_{i_1}^-.1;\;\; X_{i_3}^+ = X_{i_1}^-.1;\;\; X_i^- = X_{i_2}^- \sqcup X_{i_3}^-$$

그림 10 상수 및 이명 분석을 위한 방정식 생성 규칙

행 결과로부터 얻어진다.

if $e_{i_1}\ e_{i_2}\ e_{i_3}$는 조건 부분의 식이 먼저 계산된 후에 두 식 중의 하나가 계산되게 된다. 본 분석에서는 식의 선택 부분을 요약 하여 e_{i_2}와 e_{i_3}의 결과를 조인함으로써 전체의 결과를 생성한다.

call $e_{i_0}\ e_{i_1} \ldots e_{i_n}$은 e_{i_0}으로부터 생성된 함수값을 실인자 $e_{i_1} \ldots e_{i_n}$에 적용하는 계산을 수행한다. MIL에서는 함수를 값으로 사용하므로 e_{i_0}로부터 하나 이상의 함수값이 생성될 수 있다. 따라서 e_{i_0}에서 생성 가능한 함수값들 중 실인자의 개수가 n이하인 함수에 대하여 각 함수 몸체(body)의 실행 결과를 모두 조인하여 전체 결과를 생성한다.

$e_i = \mathrm{procedure}\ (x_1 \ldots x_n)\ e_{i_1}$은 e_{i_1}을 함수 몸체로 가지는 함수 클로저를 생성하는 식으로서 $ProcId(i)$는 식 e_i로부터 생성된 함수의 이름을 나타낸다. e_{i_1}의 수행 전 상태는 위 함수를 호출하는 모든 부분식들에 대하여 해당 함수 호출 전의 변수 환경에 실인자와 형식인자의 바인딩(binding)을 추가한 변수 환경을 각각 생성하고 이들을 모두 조인함으로써 얻어진다.

그림 10의 식들을 보면 증가분 기반 계산이 계산을 절약할 수 있는 부분을 발견할 수 있다. 대부분의 최적화는 mapjoin 식의 계산에서 이루어진다.

read 식의 경우 mapjoin $(\lambda x.(\mathrm{ap}(X_{i_1}^-.1)x)\ ((X_{i_1}^-.2).1)$을 수행하는데 이는 e_{i_1}식이 계산 가능한 모든 메모리 위치에 대하여 해당 메모리의 값을 읽어낸 후에 그 결과를 모두 조인하는 것을 말한다. 이때 만약 이 read문의 반복된 계산에서 메모리의 상태는 이전 계산과 똑같고 $((X_{i_1}^-.2).1)$로부터 계산되는 메모리 위치들의 집합만 증가했다면 증가분 계산에서는 새로운 메모리 위치만 읽어 증가분을 계산하게 되는데 반해 단순한 반복 계산에서는 이전에 읽어 들인 값들도 다시 읽어 들여 값들을 모두 조인해야 하므로 더 많은 계산을 수행하게 된다. write 식에서도 증가분 계산에서 저장할 값의 증가분과 저장할 위치의 증가분만을 고려하여 연산의 결과로 생성되는 메모리 상태의 증가분을 계산하므로 계산의 양을 줄일 수 있다.

가장 많은 절약은 메모리 상태의 조인 연산으로부터 얻어진다. procedure 문에서 함수 몸체의 수행 전 상태의 계산은 증가분 계산을 통해 가장 많은 계산을 절약할 수 있는 부분이다. 함수 몸체의 수행 전 메모리 상태의 증가에 영향을 미치는 것으로는 함수 호출시의 변수 환경의 증가, 함수 호출시 실인자 값의 증가 및 새로운 함수 호출의 등장 등을 들 수 있다. 그런데 증가분 기반 계산에서는 복수개의 함수 호출 부분들 중 함수 몸체 계산 이전의 메모리 상태 증가가 발생한 부분에 한해서

만 조인을 해주게 되므로 메모리 상태의 조인을 상당히 줄일 수 있다. call 문의 수행 결과 계산 역시 해당 함수 호출과 관련된 함수 몸체의 수행 후 상태들 중 증가한 것들만을 조인하므로 메모리 상태의 조인 연산을 절약하게 된다.

6. 구현 및 실험

본 논문에서 제안된 증가분 기반 고정점 생성 알고리즘의 성능을 실험하기 위하여 제안된 방법을 요약 해석에 기반한 정적 분석 시스템인 Z1에 적용하였다[14]. 그림 11은 정적 분석 시스템의 실행 구조를 보여준다. 구현되는 정적 분석은 요약 해석 공간과 입력 언어의 요약된 의미를 기술함으로써 정의되며 이로부터 방정식 생성기(equation generator)가 생성된다. 방정식 생성기는 분석이 대상이 되는 MIL 프로그램을 입력으로 받아 프로그램 내의 각각의 부분식들 간의 관계를 나타내는 방정식을 생성하며, 방정식의 해를 고정점 생성 방법을 통해 계산함으로써 분석 결과를 얻게 된다. 증가분 기반 고정점 계산의 효과를 측정하기 위하여 일반적인 고정점 계산 엔진도 구현하였다.

실험을 위한 정적 분석으로는 메모리 생존 시간 분석

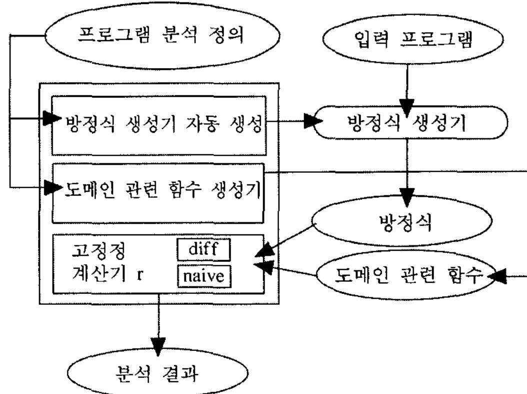

그림 11 정적 분석 시스템의 동작 구조

(memory life time analysis)과 이전 절에서 설명한 상수 및 이명 분석을 구현하였다. 메모리 생존 시간 분석은 각각의 메모리 위치에 대하여 해당 메모리의 할당 시점과 마지막 사용 시점을 분석하는 것으로서, 각각의 시점을 요약하기 위하여 프로시져의 호출 및 복귀(return) 기록을 사용한다[14].

실험 결과는 표 1과 같다. 입력 프로그램들은 시뮬레이션과 수치 계산을 위한 C와 포트란 프로그램들이다. "naive"는 그림 1의 일반적인 워크리스트 알고리즘을 사용한 경우, "diff"는 그림 3의 증가분 계산에 기반 한 워크리스트 알고리즘을 사용한 경우의 고정점 계산 소요 시간을 나타내며, 제시된 워크리스트 알고리즘은 임의의 스케쥴링 방법을 사용할 수 있으므로, FIFO(First In First Out)와 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.

대부분의 프로그램에서 LIFO 스케쥴링과 증가분 계산 방법을 사용한 경우에 가장 적은 계산 시간을 사용하였다. 증가분 계산의 경우에는 LIFO 스케쥴링 방법을 사용한 경우에 대부분 성능이 향상됨을 볼 수 있는데 이는 증가분 기반 계산에 있어서도 워크리스트 스케쥴링이 성능에 중요한 영향을 미침을 나타낸다. 메모리 생존시간 분석의 FIFO 스케쥴링과 wator와 gauss1 프로그램의 경우 증가분 기반 계산을 함으로써 분석시간이 증가하였는데, 이는 새로운 증가분을 이전의 결과와 합쳐주는 계산으로 인한 부담이 증가분만을 계산함으로써 얻어지는 계산양의 감소보다 더 컸기 때문으로 판단된다.

7. 결 론

본 연구에서는 단조 증가 함수의 고정점 계산 속도를 높이기 위하여 증가분에 기반한 고정점 계산 방법을 제시하였다. 증가분에 기반한 고정점 계산은 반복된 함수의 적용 시에 이전 결과에 그대로 함수를 적용하는 것이 아니라 인자값의 증가로 인한 결과값의 증가분만을

표 1 상수 및 이명 분석과 메모리 생존 시간 분석을 위한 고정점 생성 시간

분석	상수 및 이명 분석				메모리 생존 시간 분석			
스케쥴링	FIFO		LIFO		FIFO		LIFO	
워크리스트 알고리즘	naive	diff	naive	diff	naive	diff	naive	diff
abmoeba	44.16	16.11	74.14	3.39	2892.3	2966.1	2816.1	1860.5
simplex	120.14	49.13	296.12	7.41	5596.1	6009.5	7444.5	4219.6
gauss	34.42	20.14	34.51	1.51	2364.1	2575.1	2503.4	1465.3
TIS	12.48	10.08	4.00	1.43	4093.1	4455.4	369.5	348.39
gauss1	9.25	11.57	0.24	0.26	170.0	209.6	1.39	2.13
wator	59.20	95.04	4.21	2.06	1920.2	2146.3	650.3	692.5

계산하도록 함으로써 고정점 계산에 소요되는 계산의 양을 줄이게 된다. 또한 다양한 워크리스트 스케줄링을 적용할 수 있는 증가분 기반 워크리스트 알고리즘을 제시하였다.

본 연구의 방법을 요약 해석에 기반한 상수 및 이명 분석과 메모리 생존 시간 분석에 적용해 본 결과, 증가분 계산 방법을 통하여 전체 계산량을 단축할 수 있으며 증가분 계산에도 적절한 워크 리스트 스케줄링 방법의 사용이 중요함을 확인할 수 있었다.

향후 연구는 다음과 같다. 우선 분석 속도의 저하를 보인 입력 프로그램들을 조사하여, 제시된 방법의 부담을 정확히 분석해야 하며, 이를 극복할 수 있는 방법을 찾아내는 것이 필요하다. 또한 제시된 워크리스트 알고리즘은 다양한 스케줄링 방법을 적용할 수 있으므로, 증가분 기반 계산에 적합한 워크리스트 스케줄링 방법을 개발하는 것이 필요하다.

참 고 문 헌

[1] 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.

[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, pp.296-306, 1986.

[3] 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, pp.31-68, 1995.

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

[5] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman, Compilers, principles, techniques, and tools, Addison Wesley, 1988.

[6] Robert Paige and Shaye Koenig, "Finite differencing of computable expressions," ACM Trans. on Programming Languages and Systems, Vol.4, No.3, pp.402-454, 1982.

[7] Yanhong A. Liu, "Efficiency by incrementalization : an introduction," Higher-Order and Symbolic Computation, Vol.13, No.4, pp.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, pp.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), pp.90-104. LNCS 1381, Springer Verlag, 1998.

[10] Christian Fecht and Helmut Seidl, "A faster solver for general systems of equations," Science of Computer Programming, Vol.35, No. 2, pp.137-161, 1999.

[11] Joonseon Ahn, "A Differential Evaluation of Fixpoint Iterations," The Second Asian Workshop on Programming Languages and Systems, Taejon, Korea, Dec., 2001.

[12] 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, p.238-252, 1977.

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

[14] Kwangkeun Yi, Automatic Generation and Management of Program Analyses, Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1993.

[15] 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.

안 준 선

1988년 3월~1992년 2월 서울대학교 계산통계학과(학사). 1992년 3월~1994년 2월 KAIST 전산학과(석사). 1994년 3월~2000년 8월 KAIST 전자전산학과 전산학전공(박사). 2000년 9월~2001년 8월 KAIST 프로그램분석시스템연구단(ROPAS) 연구원. 2001년 9월~현재 한국항공대학교 항공전자 및 정보통신공학부 조교수. 관심분야는 컴파일러, 정적분석, 정보 검색, 프로그램 보안, 임베디드시스템 등