■ 2007년도 학생논문 경진대회 수상작

함수 요약을 이용한 모듈단위 포인터분석
(A Modular Pointer Analysis using Function Summaries)

박 상 운 † 강 현 구 †† 한 태 숙 †††
(Sangwoon Park) (Hyun-Goo Kang) (Taisook Han)

요 약 본 논문에서는 업데이트 기록에 기반한 모듈단위 포인터 분석 알고리즘을 제안한다. 여기서 모듈이란 상호 재귀적인 함수들의 집합을 의미하며, 모듈단위 분석이란 한 모듈을 분석 시에 다른 모듈의 소스코드가 필요하지 않는 분석을 의미한다. 일반적으로 이러한 형태의 분석은 분석 대상 모듈의 호출 문맥을 알 수 없는 상태에서 분석을 수행하여야 하기 때문에, 프로그램의 흐름 또는 호출 문맥에 관련하여 분석의 정확도를 잃을 수 있다. 본 논문에서는 업데이트 기록이라 이름 지어진 모듈단위 분석 공간을 고안하여, 프로그램 문맥과 흐름에 민감한 정확도를 가지는 모듈단위 포인터 분석 방법을 제안한다. 업데이트 기록은 함수의 호출 문맥에 독립적으로 메모리 상태를 요약할 수 있을 뿐만 아니라, 메모리 반응이 일어난 순서에 관한 정보를 유지할 수 있다. 업데이트 기록의 이러한 특성은 모듈단위 분석을 정형화 하는데 효과적으로 사용되었을 뿐만 아니라, 분석의 정확도를 높이기 위해 죽은 메모리 반응 또는 관련된 별칭 문맥을 구분하는 데에도 효과적으로 사용될 수 있었다.

키워드 : 포인터분석, 모듈단위 분석, 상향방식 분석, 실행 요약

Abstract In this paper, we present a modular pointer analysis algorithm based on the update history. We use the term 'module' to mean a set of mutually recursive procedures and the term 'modular analysis' to mean a program analysis that does not need the source codes of the other modules to analyze a module. Since a modular pointer analysis does not utilize any information on the callers, it is difficult to design a precise analysis that does not lose the information related to the program flow or the calling context. In this paper, we propose a modular and flow- and context-sensitive pointer analysis algorithm based on the update history that can abstract memory states of a procedure independently of the information on the calling context and keep the information on the order of side effects performed. Such a memory representation not only enables the analysis to be formalized as a modular analysis, but also helps the analysis to effectively identify killed side effects and relevant alias contexts.

Key words : pointer analysis, modular analysis, bottom-up analysis, summary

· 본 연구는 지식경제부 및 정보통신연구진흥원의 대학 IT연구센터 지원 사업의 연구결과로 수행되었음(IITA-2008-C1090-0801-0020)

† 학생회원 : 한국과학기술원 전자전산학과
inverse.midas@gmail.com
†† 학생회원 : 한국과학기술원 전산학과
hgkang@ropas.kaist.ac.kr
††† 종신회원 : 한국과학기술원 전자전산학과 교수
han@cs.kaist.ac.kr

논문접수 : 2007년 7월 12일
심사완료 : 2008년 9월 10일

1. 서 론

1.1 프로그램 분석 방식

보통 프로그램 분석은 하향방식(top-down)으로 설계 및 구현된다. 하향방식 프로그램 분석이란 프로그램의 시작점(예, main함수)부터 분석을 시작하여 함수 호출을 만날 때마다, 호출된 함수의 해당 함수의 소스코드를 이용하여 분석을 계속해 나가는 분석방법을 의미한다. 따라서 분석이 동작하기 위해서는 프로그램 상에서 호출하는 모든 함수의 소스 코드(즉, 전체 프로그램)가 필요하다.

상향방식(bottom-up) 프로그램 분석은 하향방식과 반

대로 프로그램을 함수호출 그래프의 말단에 해당하는 함수들부터 분석해 나가는 방식이다. 상향방식 프로그램 분석은 함수를 분석할 때 함수의 호출문맥에 대한 정보가 전혀 없이 분석을 수행하여 그 함수의 실행요약(summary)을 작성한다. 차후 그 함수를 호출하는 함수를 분석할 때에는 해당 함수의 소스 코드가 아니라 미리 분석해둔 결과인 실행요약을 사용하여 함수 호출을 분석한다.

상향방식 분석 방법은 실용적인 면에서 여러 가지 장점들이 있다. 첫 번째로, 상향방식 분석은 상위 모듈(caller) 없이 하위 모듈(callee)들에 대한 분석을 수행할 수 있기 때문에, 라이브러리와 같이 프로그램의 시작점이 없는 완전하지 않은 프로그램도 분석 할 수 있다. 따라서, 상향방식 분석은 각각의 소프트웨어 부품들이 따로 개발되고 차후 의도한 순서로 합치는 식으로 개발되는 모듈단위 소프트웨어 개발 환경에서 개발 도중 활용될 수 있다. 두 번째로, 상향방식 분석은 전체 프로그램과 이에 대한 분석 결과를 분석 중 동시에 메모리에 둘 필요가 없기 때문에 하향방식 분석보다 종종 더 큰 프로그램을 분석할 수 있다. 또한, 실행요약을 사용하여 함수 호출을 분석 하기 때문에 함수가 호출될 때 마다 코드의 재분석을 기본으로 하는 하향방식 프로그램 분석 보다 종종 더 빠르게 동작한다. 마지막으로, 코드가 변경 되었을 때 해당 변경에 관련이 없는 부분들에 대해서는 다시 분석을 할 필요 없이 기존의 분석 결과를 재사용 할 수 있다.

1.2 연구 목표 및 핵심 아이디어

C 프로그램을 분석하는데 있어 포인터 분석은 필수적이다. 우리의 연구 목표는 상향방식으로 개발되는 보통의 모듈단위 프로그래밍 환경하에서 정확도를 많이 잃지 않으면서 효과적으로 동작할 수 있는 상향방식의 포인터분석을 설계하는 것이다. 특히, 아래 코드에서 설명된 두 가지 문제를 잘 다루려고 한다.

```
f (int **p, int **q, int **r) {
 l1: *p = &g1; l2: *q = &g2; l3: r* = &g3; l4:
   return *q;
}
main () {
 int *i, *j, *k;
 l5: f(&i,&j,&j); l6: f(&j,&k,&i);
}
```

아래에서 우리는 함수 f의 시작점에서 프로그램 식 &g1, &g2, &g3, &p, &q, &r, p, q, r이 나타내는 메모리 주소를 각각 g1, g2, g3, p, q, r, p*, q*, r*이라 명칭할 것이다.

강한 업데이트: 함수 호출 l5와 같이 q와 r이 같은 메모리 주소를 가리키게 되는(즉, q*과 r*이 같은 경우) 호출문맥 하에서는 함수 내에서 할당문 l3가 할당문 l2의 메모리 반응(효과)을 지운다. 이 경우 l2의 메모리 반응은 죽은 메모리 반응(killed side effect)이라 한다. 분석기는 죽은 메모리 반응을 안전하게 무시 할 수 있으며 이런 방식의 메모리 갱신을 이를 강한 업데이트(strong update)라 한다[1]. 반면 l6와 같이 p, q, r이 다른 메모리 주소를 가리키게 되는 호출문맥 하에서는 할당문 l1, l2, l3의 메모리 반응을 모두 유지하며 분석을 수행해야 한다. 따라서, 상향 방식의 프로그램 분석을 정확하게 수행하기 위해서는 이와 같은 호출 문맥의 차이에 따른 함수 f의 메모리 반응의 차이를 구분할 수 있어야 한다.

별칭 문맥 민감성: 임의의 함수 내에서 하나의 메모리 주소는 다른 이름으로 명칭 할 수 있다. 위의 예제에서 메모리 주소 q*과 r*은 함수 내에서 다른 이름으로 명칭 되었으나, 호출 문맥에 따라 실제로는 같은 메모리 주소를 가리킬 수도(예, 함수호출 l5) 다른 메모리 주소를 가리킬 수도(예, 함수 호출 l6) 있다. 우리는 이러한 호출 문맥의 차이를 별칭 문맥(alias context)이라 부르는데 함수의 동작(behavior)은 이러한 별칭 문맥의 차이에 따라 달라질 수 있다. 예를 들어 함수 f의 리턴 값은 함수호출 l5에서는 g2가 되며, 함수호출 l6에서는 g3가 된다. 따라서, 상향 방식의 프로그램 분석을 정확하게 수행하기 위해서는 l5에서는 p*이 i이고 l6에서는 j이라는 보통의 호출문맥의 차이뿐만 아니고, q*과 r*이 같은지 여부를 뜻하는 별칭 문맥의 차이에 따른 함수 실행의 차이를 추적할 수 있어야 한다.

우리의 상향 방식 분석은 만약 메모리 반응의 순서에 관한 정보가 있다면, 위에 설명된 죽은 메모리 반응과 관련 별칭 문맥을 효과적으로 구분해 낼 수 있다는 관찰에 기반하고 있다. 예를 들어, 함수 f의 실행요약이 [p*→{g1}][q*→{g2}][r*→{g3}] 식으로 q*이 r*보다 먼저 업데이트 되었다는 정보를 가지고 있다면, 차후 함수가 호출 되어 실행요약을 적용할 때, l5와 같이 q*과 r*이 각각 j, j로 같은 메모리 주소가 되도록 함수 f가 호출되었을 때에는 q*에 대한 메모리 반응을 지우고, l6와 같이 다르면 q*에 대한 메모리 반응을 유지하는 방식으로 분석을 수행할 수 있다. 이번엔 별칭 문맥의 차이에 따른 함수 f의 리턴 값의 변화를 추적해 보자. 앞서 설명했듯이 함수의 리턴 값은 q*과 r*이 같은 주소인지 여부에 따라 l5에서는 g2로 l6에서는 g3로 달라지게 되는데 우리의 분석에서는 위와 같은 형태의 업데이트 기록을 이용하여 메모리의 상태를 요약하기 때문에, l4에서

표현식 *q의 값을 계산할 때 r*과 q*이 같을 때는 g3가 r*과 q*이 다를 때는 g2가 리턴 값이 달라진다는 점을 구분해낼 수 있게 된다. 따라서, 위의 두 별칭 문맥의 차이에 따른 리턴 값의 차이를 구분하여 함수의 실행요약을 만들어 낸다. 참고로 위 함수에 대해 가능한 모든 별칭 문맥의 수는 매우 많음에 주지하자. 우리의 분석은 관련연구[20]처럼 이러한 모든 별칭 문맥을 모두 고려하는 것이 아니라 필요한 별칭 문맥만을 도입한다. 또한, 예제에서 p*과 q*간의 별칭 관계는 함수의 리턴 값에 영향을 미치지 않음도 주지하자. 우리의 분석은 q*과 r* 간의 별칭 관계와 같이 실행에 영향을 주는 관련 별칭 문맥만을 분석 중 도입하여 함수의 실행을 요약한다.

본 논문에서는 위에서 직관적으로 설명된 업데이트 기록을 호출 문맥에 상관없이 함수의 메모리 상태를 요약할 수 있는 정형화된 프로그램 분석 공간으로 정의하고, 이를 활용하여 위에서 설명된 정도의 정확도를 가지는 상향방식 포인터 분석을 정형화 하였다. 따라서, 기존의 포인터 분석 연구들에서 충분히 논의 되지 않았던, 안전성 증명과 알고리즘이 끝남 증명과 같은 정형화된 추론(formal reasoning) 또한 가능하다는 장점 또한 가지고 있다.

1.3 관련 연구

대부분의 프로그램 흐름 문맥과 호출 문맥에 민감한 포인터 분석들은 하향 방식으로[2-13] 설계되었다. 하지만 이러한 포인터 분석들은 상향 방식의 분석에 비해 전체 프로그램이 주어졌을 때에만 동작 할 수 있기 때문에 본 연구의 목표 환경인 프로그램의 개발 도중 사용할 수 없다는 단점이 있다. 반면 전체 프로그램이 주어진 상태에서 하향 방식으로 분석을 수행할 수 있다면, 일반적으로 1.2절에서 언급한 문제들이 발생 하지 않아 더 자세한 분석을 수행할 수 있는 장점이 있다.

상향 방식으로 포인터 분석을 수행하는 대부분의 기존 연구들은 프로그램의 흐름을 무시하는 방식으로[14-19] 설계되었다. 이러한 분석 방식은 보통 함수의 요약을 만들어 내는 방법이 상대적으로 간단하기 때문에 성능이 좋은 장점이 있다. 하지만, 프로그램의 흐름에 민감한 프로그램의 성질을 분석하려 할 때에는 정확도가 너무 떨어진다는 단점이 있다, 예를 들어, 아래의 프로그램을 생각해 보자.

```
1: int i, *p = NULL;
2: p = &i;
3: *p = 10;
```

이와 같은 프로그램에 대해서는 라인 2에서 강한 업데이트를 수행할 수 있는 프로그램의 흐름에 민감한 분석만이 라인 3에서 빈 포인터 참조 에러가 나지 않는다는 것을 증명할 수 있다.

한편, 프로그램 흐름과 호출 문맥에 민감하면서도, 상향 방식으로 포인터 분석을 수행하는 기존 연구들도 있다[20-25]. 이중에서 [23]은 이 본 논문과 마찬가지로 *x = &i; 와 같은 간접 할당문에서도 강한 업데이트를 수행할 수 있으며, 함수를 분석할 때 관련된 별칭 문맥들만을 찾아가면서 분석을 수행할 수 있다. 하지만 [23]에서 사용 하는 분석 공간(points-to set)은 메모리 업데이트 순서를 드러내지 않는다. 따라서, 강한 업데이트 처리와, 관련 별칭 문맥을 구분을 위해 별도의 방법과 휴리스틱이 필요 했다. 또한, 이 방법으로는 1.2절 예제에서 논의된 함수 호출 시 강한 업데이트를 수행 할 수 없으며, 관련 별칭 문맥을 찾을 때 위 예제의 p*과 q* 간의 별칭 관계처럼 필요 없는 별칭 문맥을 도입할 수도 있는 단점이 있었다. [24]는 본 논문의 접근 방법과 마찬가지로 함수를 분석할 때 미리 결정할 수 없는 업데이트들이 있으면 그래프 형태로 이러한 업데이트들의 순서를 기억할 수 있는 함수의 실행요약을 작성한다. 따라서, 함수 호출 시 미루어둔 업데이트들에 대해 강한 업데이트를 수행할 수 있다. 한편 본 논문에서 제안하는 분석은 메모리 읽기에 대해서 항상 미리 계산을 수행한다. 이에 따라, 관련된 별칭 문맥을 찾아서 그에 대한 수행 결과를 미리 예측하여 실행요약을 작성한다. 반면 [24]는 메모리 읽기에 대해서도 미리 계산을 수행할 수 없을 경우 계산을 미루기 때문에 관련된 별칭 문맥에 따른 프로그램 수행의 차이를 미리 예측하지 않는다. 또한, 본 논문은 분석 알고리즘의 안전성 및 끝남에 대해 논할 수 있도록(formal reasoning) 분석이 정형화 되었다는 점에서 [20-24]와 다르다. [25]는 호출 문맥 없이 주어진 함수가 포인터 관련 오류 없이 잘 동작할 수 있는 조건을 찾아주며 함수의 실행요약을 작성한다. 하지만, 분석의 대상이 프로그램의 각 표현식이 가리킬 수 있는 메모리 주소를 계산해주는 포인터 분석이 아니고 메모리의 모양을 예측하는 분석이다.

2. 대상 언어

우리가 대상으로 하는 언어의 문법은 그림 1과 같다. 우리는 포인터에 대해서만 분석을 할 것이기 때문에, C 언어의 구문들 중에서 포인터와 관련된 핵심 부분만을 선택하였다. if문의 경우, 조건 부분이 빠져있는 것은 우리가 포인터 분석 시에 조건 부분을 활용하지 않을 것이기 때문이다. 또한 재귀함수로 표현될 수 있는 while 문은 생략되었다. 함수는 매개변수들, 지역변수들, 그리고 몸통으로 이루어진다. 함수의 반환 값이 없는 이유는, $f(\&x)$와 같이 포인터를 이용해서 받을 수 있기 때문이다. 프로그램의 시작점이 존재하지 않는 라이브러리에 대해서도 분석을 원하기 때문에, 프로그램은 함수들의

$$
\begin{array}{llll}
y & \in & FormalVar & \\
z & \in & LocalVar & \\
x & \in & Var & = FormalVar \cup LocalVar \\
f,g & \in & Id & \\
e & \in & Expression & ::= \&x \mid *e \mid \text{null} \\
s & \in & Statement & ::= *e := e \mid f(\overline{e}) \mid s;s \mid \text{if}(s,s) \mid \text{skip} \\
c & \in & Closure & = FormalVar^* \times LocalVar^* \times Statement \\
p & \in & Program & = Id \xrightarrow{fin} Closure
\end{array}
$$

그림 1 문법

집합으로 정의된다. x와 같은 *Expression*이나 $x := e$ 와 같은 문법이 존재하지 않는데, 이것은 각각 $*(\&x)$과 $*(\&x) := e$ 로 나타낼 수 있기 때문이다.

위 언어의 의미공간(semantic domain)은 그림 2에 정의되어 있다. 값 (*Value*)은 변수 (*Var*)에 $\perp$을 추가한 펼쳐진 의미공간이다. 이것은 곧 주소를 나타내며, 여기서 $\perp$은 아무것도 가리키지 않음을 나타낸다. 정수형 자료는 분석하지 않기 때문에 값 의미공간에 포함하지 않는다. 환경(*Env*)은 각각의 함수의 입력 값과 메모리에 대한 함수의 수행결과 메모리들의 집합이다. 이렇게 이미 요약된 의미구조를 정의하는 이유는, if문에서 then 방향과 else 방향의 실행결과를 모두 포섭하여 모으기 때문이다.

그림 2는 이 언어의 의미를 뭉친 의미구조(collecting semantics)를 사용하여 정의하고 있다.

표현식(*Expression*)은 메모리 반응이 없으므로 주어진 메모리를 변화시키지 않고 값만을 계산한다. $\&x$는 x의 주소를 뜻한다. $*e$는 먼저 e가 나타내는 값(즉 변수들의 집합)을 계산한 후에, 각각의 변수의 값을 주어진 메모리에서 읽은 후 그것들을 모두 모은다.

명령문(*Statement*)은 주어진 메모리 집합 (2^{Mem})에 대해 변경된 메모리들을 계산한다. $*e_1 := e_2$는 각각의 메모리에 대해, 먼저 e_1을 계산함으로써 $*e_1$이 나타내는 변수들을 알아내고, 그 각각의 변수들에 대해 e_2가 나타

[Semantic Domain]

$$
\begin{array}{rcll}
v & \in & Value & = Var_\perp \\
m & \in & Mem & = Var \xrightarrow{fin} Value \\
\rho & \in & Env & = Id \xrightarrow{fin} 2^{Value^* \times Mem} \to 2^{Mem}
\end{array}
$$

[Memory Operations]

$$
\begin{array}{rcl}
m[x_1 \mapsto v] & = & \lambda x.\text{if } x = x_1 \text{ then } v \text{ otherwise } m(x) \\
m\backslash\{\overline{x_i}\} & = & \lambda x \in \text{dom}(m)\backslash\{\overline{x_i}\}.\text{if } m(x) \in \{\overline{x_i}\} \text{ then } \perp \text{ else } m(x) \\
M\backslash\{\overline{x_i}\} & = & \{m\backslash\{\overline{x_i}\} \mid m \in M\}
\end{array}
$$

[Semantics]

$$
\begin{array}{rcl}
\mathbf{E}[\cdot] & : & Expression \to Mem \to Value \\
\mathbf{E}[\![\&x]\!]\ m & = & x \\
\mathbf{E}[\![*e]\!]\ m & = & \text{let } v = \mathbf{E}[\![e]\!]\ m \text{ in if } v = \perp \text{ then } \perp \text{ else } m(v) \\
\mathbf{E}[\![\text{null}]\!]\ m & = & \perp \\
\\
\mathbf{S}[\cdot] & : & Statement \to Env \to 2^{Mem} \to 2^{Mem} \\
\mathbf{S}[\![*e_1 := e_2]\!]\ \rho\ M & = & \{m[v_1 \mapsto v_2] \mid m \in M,\ v_1 = \mathbf{E}[\![e_1]\!]\ m,\ v_1 \neq \perp,\ v_2 = \mathbf{E}[\![e_2]\!]\ m\} \\
\mathbf{S}[\![f(\overline{e_i})]\!]\ \rho\ M & = & \rho\ f\ \{(\overline{(\mathbf{E}[\![e_i]\!]\ m)},\ m) \mid m \in M\} \\
\mathbf{S}[\![s_1;s_2]\!]\ \rho\ M & = & \mathbf{S}[\![s_1]\!]\ \rho\ (\mathbf{S}[\![s_2]\!]\ \rho\ M) \\
\mathbf{S}[\![\text{if}(s_1,s_2)]\!]\ \rho\ M & = & (\mathbf{S}[\![s_1]\!]\ \rho\ M) \cup (\mathbf{S}[\![s_2]\!]\ \rho\ M) \\
\mathbf{S}[\![\text{skip}]\!]\ \rho\ M & = & M \\
\\
\mathbf{C}[\cdot] & : & Closure \to Env \to 2^{Value^* \times Mem} \to 2^{Mem} \\
\mathbf{C}[\![(\overline{y_i},\overline{z_j},s)]\!]\ \rho\ VM & = & \text{let } \overline{y_i'},\ \overline{z_j'} \text{ be fresh new variables not occuring in } VM \\
 & & (\mathbf{S}[\![s[\overline{y_i'}/\overline{y_i}][\overline{z_j'}/\overline{z_j}]]\!]\ \rho\ \{m[\overline{y_i'} \mapsto \overline{v_i}][\overline{z_j'} \mapsto \perp] \mid (\overline{v_i}, m) \in VM\})\backslash\{\overline{y_i'},\overline{z_j'}\} \\
\\
\mathbf{P}[\cdot] & : & Program \to Env \\
\mathbf{P}[\![p]\!] & = & \text{lfp } \lambda\rho.\lambda f \in \text{dom}(p).\mathbf{C}[\![p(f)]\!]\ \rho
\end{array}
$$

그림 2 의미구조

내는 값으로 메모리를 갱신한다. 이 각각의 경우를 모두 포섭해야 하므로, 결과로써 나오는 모든 메모리들을 모은다. 참고로, $m[x_1 \rightarrow v]$는 주어진 메모리 m에서 x의 값을 v로 업데이트한 메모리를 나타내고, $m/\{\overline{x_i}\}$는 메모리 m의 정의역과 치역에서 $\overline{x_i}$들이 제거된 메모리를 뜻한다. $f(\overline{e_i})$은 주어진 환경하에서 주어진 각각의 입력 메모리들 하에서 계산된 인자 값들을 사용하여 함수의 몸통을 수행한 결과 메모리들의 집합을 얻는다. if(s_1, s_2)는 조건 부분이 없기 때문에, 항상 두 경우(then과 else의 가능성을 모두 포섭하도록, 각각의 결과를 집합으로 모은다.

클로저(*Closure*)는 주어진 환경하에서, 인자(*Value**)와 메모리의 짝들을 받아서, 새로운 메모리들을 내놓는 함수값을 뜻한다. 이때, 몸체에 나타나는 매개변수들 $(\overline{y_i})$과 지역변수들 $(\overline{z_i})$을 주어진 메모리에 나타나지 않은 새로운 변수들로 치환시키는 것은 바로 재귀함수를 허용하기 때문이다. 즉, 함수의 스택(stack)의 값을 추적하기 위해서이다. 결과로 나오는 메모리에서 $\overline{y_i}$와 $\overline{z_i}$를 제거하는 것도 같은 이유에서이다. 한편 주어진 환경은 몸체에서 함수호출이 있을 경우 사용된다.

마지막으로 프로그램(*Program*)의 의미는 바로 프로그램에 있는 모든 함수들의 의미의 집합을 뜻한다.재귀함수 및 상호재귀함수들의 의미는 위와 같이 최소고정점(least fixpoint)으로 정의된다.

3. 분석 알고리즘

3.1 접근 경로(Access Path)

상향방식의 프로그램 분석을 위해서는 함수 내에서 접근 가능한 메모리 주소들을 호출 문맥에 상관 없이 이름 짓고, 차후 함수가 호출이 되면 호출 함수의 견지에서 해당 이름들을 다시 복구 할 수 있는 방안이 필요하다(parameterization). 본 절에서는 잘 알려진 접근 경로의 개념[3]을 사용하여 함수 내에서 접근 가능한 모든 메모리 주소를, 호출 문맥에 상관없이 유한하게 이름 짓는다. 접근 경로는 함수 내에서 접근 가능한 모든 메모리 주소를, 함수의 변수들로부터 접근 하는 방법 자체로 이름 짓는 방법이고 정형화된 정의는 그림 3에 나타나있다.

확장 접근 경로(*ExtendedAccess*)는 인자 값으로부터 한 번 이상 참조(dereference)된 메모리 위치들을 나타낸다. 예를 들어 y가 어떤 함수의 매개변수일 때, $y*$는 그 함수가 불리는 시점에서 y가 가리키는 메모리 위치를 뜻한다. 마찬가지로 $y**$은 $y*$이 함수가 불리는 시점에서 가리키는 메모리 위치를 뜻한다. 앞으로 $(y, *^m)$을 $y*^m$으로 표기하기로 하자.

접근 경로(*AccessPath*)는 확장 접근 경로뿐만 아니라, 지역변수나 매개변수까지도 포함한다. 함수가 호출되고 몸통 수행 후 다시 되돌아가게 되면, 지역변수와 매개변수는 스택 변수이므로 확장 접근 경로와는 달리 그들에 대한 메모리 갱신들이 아무런 의미가 없어진다는 점 때문에, 우리는 확장 접근 경로를 따로 구분한다.

일반적으로 하나의 함수에서 접근 가능한 메모리 주소는 무한하다. 예를 들어 매개변수 y를 이용하여 접근 가능한 메모리주소는 y, $y*$, $y**$, $y***$, … 식으로 무한히 존재할 수 있다. 따라서, 하나의 함수를 분석할 때 그 함수가 가질 수 있는 접근 경로들을 유한 공간으로 요약하는 과정이 필요하다. 그림 3에 정의된 함수 α는 임의의 요약되지 않는 접근 경로를 받아 요약된 접근 경로를 계산해 주는 함수이다. α는 주어진 접근 경로 ap가 확장 접근 경로일 경우 α_{eap}를 이용하여 요약하고 그렇지 않으면 (매개변수 혹은 지역변수일 경우) 그대로 둔다. κ(x)=n는 매개변수 x에서 출발하는 메모리 참조

$$
\begin{array}{lcll}
eap & \in & ExtendedAccessPath & = & FormalVar \times \{*\}^+ \\
ap & \in & AccessPath & = & ExtendedAccessPath \cup LocalVar \cup FormalVar \\
 & & & & \\
 & & \kappa & : & FormalVar \rightarrow \mathbb{N}\backslash\{0\} \\
 & & \omega & : & FormalVar \rightarrow \mathbb{N}\backslash\{0\} \\
 & & \alpha_{eap} & : & ExtendedAccessPath \rightarrow ExtendedAccessPath \\
 & & \alpha_{eap}(y*^m) & = & y*^n \text{ where} \\
 & & & & \text{if } m \leq \kappa(y) \text{ then } n = m \\
 & & & & \text{else } n = \kappa(y) + ((m - \kappa(y)) \bmod \omega(y)) \\
 & & & & \\
 & & \alpha & : & AccessPath \rightarrow AccessPath \\
 & & \alpha(ap) & = & \text{if } ap \in ExtendedAccessPath \text{ then } \alpha_{eap}(ap) \text{ else } ap \\
 & & & & \\
AP & \in & \widehat{AccessPath} & = & \text{ran } \alpha \\
EAP & \in & \widehat{ExtendedAccessPath} & = & \widehat{AccessPath} \cap ExtendedAccessPath
\end{array}
$$

그림 3 접근 경로

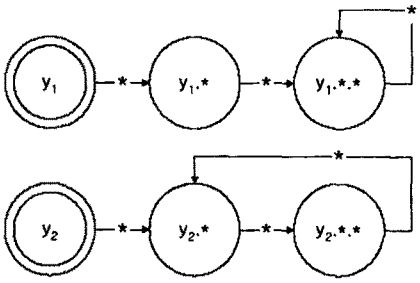

그림 4 추상화 그래프

의 깊이를 n개까지는 요약없이 추적할 것임을 나타내고, $\omega(x) = n$은 요약이 시작되는 시점에서부터, 메모리 참조들을 n개로 그룹화 하기 위해 사용한다. 참고로 그림 4는 $\kappa(y_1)=2$, $\omega(y_1)=1$, $\kappa(y_2)=1$, $\omega(y_2)=2$인 접근 경로 요약함수 α_{eap}를 결정적 유한 오토마타의 모양으로 도식화한 모습이다. 이것에 따라 요약의 예를 들면 $\alpha_{eap}(y_2****) = y_2**$이다.

이제 요약된 접근 경로 $\widehat{AccessPath}$는 α의 치역 (range), 즉, 유한하게 요약된 접근 경로의 집합으로 정의된다. 마찬가지로 요약된 접근 경로 $\widehat{ExtendedAccessPath}$는 $\widehat{AccessPath}$에서 지역변수와 매개변수를 제외한 접근 경로들로 정의된다.

3.2 분석의 의미공간

그림 5와 6은 주어진 프로그램을 분석하여 실행요약들을 만드는데 사용되는 의미공간들과 그 순서(domain order) 정의를 보여준다. 순서들은 추론 규칙(inference rules)들을 이용하여 정의 하였지만, 이 추론규칙들은 계산 가능(decidable)하게 정의 되었다. 즉 임의의 $X \sqsubseteq_X X'$에 대해 참/거짓 여부를 항상 계산할 수 있다.

가정(*Assumption*)은 어떠한 함수가 호출되는 상황에서 만족하는 가정을 나타낸다. 예를 들어, $y_1* \# y_2**$는 함수가 호출되는 상황에서 y_1*가 나타내는 메모리 주소와 y_2**가 나타내는 메모리 주소가 절대 같은 메모리 위치를 나타내지 않는다는 것을 의미한다. 그림 6의 규칙 (2)는, 어떠한 함수 내에서, 둘 다 스택 변수이거나, 둘 중 하나가 스택 변수라면 서로 같은 메모리 위치를 나타낼 수 없다는 것을 의미한다. 즉, 스택 변수는 함수가 호출되는 시점에서 생성되기 때문에 확장 접근 경로와는 절대로 같은 메모리 위치를 나타낼 수 없기 때문이다. $\{b1, ..., b_i\} \in Assumptions$은 $b_1, ..., b_i$가 모두 만족된다는 것을 나타낸다(즉, $b_1 \wedge ... \wedge b_i$). 두 $B, B' \in Assumptions$의 순서는 논리적 암시(logical implication, $\Rightarrow$)를 염두에 두고 정의하였다. 즉 $B \sqsubseteq_B B'$라는 것은 B를 가정했을 때, B'의 가정을 모두 유추할 수 있다.

원래의 의미구조에서는 값은 변수의 주소이지만, 우리의 상향 방식 분석은 어떤 함수를 분석할 때 그 함수가 호출되는 정확한 상황을 모른 채 매개변수화 된 상황에서 분석하기 때문에, 변수가 아닌 $\widehat{AccessPath}$를 이용하며, 분석의 특성상 요약(approximation)이 일어나게 되므로 $2^{\widehat{AccessPath}}$가 분석기의 값($\widehat{Value}$)으로 사용되며 이들간의 순서는 자연스럽게 부분 집합 순서($\subseteq$)로 정의된다.

이제 본 논문의 핵심 분석 공간인 업데이트 기록(*History*)을 살펴보자. 3.1절에서의 접근 경로가 메모리 주소를 호출 문맥에 상관없이 요약할 수 있는 방법이라면, 업데이트 기록은 함수의 메모리 상태를 메모리 반응의 순서를 유지하면서 호출 문맥에 상관없이 요약해줄 수 있는 방법(분석 공간)이다. 업데이트 기록 ε은 함수의 시작점에서부터 메모리 반응이 전혀 없었던 메모리 상태를 요약해준다. $h \cdot [AP \rightarrow V]$는 h라는 업데이트들이 있은 후에 마지막으로 AP가 V로 업데이트 된 후의 메모리를 나타낸다.

업데이트 기록간의 순서($\sqsubseteq_h$)는 그림 6에 정의되어 있다. 이 순서의 직관적인 의미는, 만약 $B \vdash h \sqsubseteq_h h'$이 만족된다면, B를 만족시키는 임의의 호출 문맥(호출 시 메모리 상태)에서, h를 수행한 결과를, h'를 수행한 결과

b	$\in$	*Assumption*	::=	$EAP \# EAP$
B	$\in$	*Assumptions*	=	$2^{Assumption}$
V	$\in$	$\widehat{Value}$	=	$2^{\widehat{AccessPath}}$
h	$\in$	*History*	::=	$\epsilon \mid h \cdot [AP \mapsto V]$
H	$\in$	*Histories*	=	$2^{History}$
d	$\in$	*AbstractStatement*	=	$2^{Assumptions \times Histories}$
β	$\in$	*Signatures*	=	$Id \xrightarrow{fin} FormalVar^*$
γ	$\in$	*Env*	=	$Id \xrightarrow{fin} AbstractStatement$
S	$\in$	*Substitution*	=	$ExtendedAccessPath \xrightarrow{fin} \widehat{Value}$
q	$\in$	$2^{Assumptions \times \widehat{Value}}$		
s	$\in$	$2^{Assumptions \times Substitution}$		
b	$\in$	$2^{Assumptions}$		

그림 5 분석 의미공간 (analysis domain)

$$(1)\quad \frac{(AP\#AP' \in B) \vee (AP'\#AP \in B)}{B \vdash AP\#AP'}$$

$$(2)\quad \frac{AP \neq AP' \wedge (AP \notin ExtendedAccessPath \vee AP' \notin ExtendedAccessPath)}{B \vdash AP\#AP'}$$

$$(3)\quad \frac{\forall (AP\#AP') \in B' : B \vdash AP\#AP'}{B \sqsubseteq_B B'}$$

$$(4)\quad \frac{V \subseteq V'}{V \sqsubseteq_V V'}$$

$$(5)\quad \frac{}{B \vdash h \sqsubseteq_h h}$$

$$(6)\quad \frac{B \vdash h \sqsubseteq_h h' \quad B \vdash h' \sqsubseteq_h h''}{B \vdash h \sqsubseteq_h h''}$$

$$(7)\quad \frac{}{B \vdash \epsilon \sqsubseteq_h \epsilon \cdot [AP \mapsto \{\alpha(AP*)\}]}$$

$$(8)\quad \frac{B \vdash AP\#AP'}{B \vdash h \cdot [AP \mapsto V][AP' \mapsto V'] \sqsubseteq_h h \cdot [AP' \mapsto V'][AP \mapsto V]}$$

$$(9)\quad \frac{}{B \vdash h \cdot [AP \mapsto V][AP' \mapsto V'] \sqsubseteq_h h \cdot [AP' \mapsto V'][AP \mapsto V \sqcup_V V']}$$

$$(10)\quad \frac{B \vdash h \sqsubseteq_h h' \quad V \sqsubseteq_V V'}{B \vdash h \cdot [AP \mapsto V] \sqsubseteq_h h' \cdot [AP \mapsto V']}$$

$$(11)\quad \frac{\forall h \in H : \exists h' \in H' : B \vdash h \sqsubseteq_h h'}{B \vdash H \sqsubseteq_H H'}$$

$$(12)\quad \frac{\forall (B', H') \in d' : \exists (B, H) \in d : B' \sqsubseteq_B B \wedge B' \vdash H \sqsubseteq_H H'}{d \sqsubseteq_d d'}$$

$$(13)\quad \frac{\forall f \in \mathrm{dom}(\gamma) : d \sqsubseteq_d d' \text{ where } (\_, d) = \gamma(f) \text{ and } (\_, d') = \gamma'(f)}{\gamma \sqsubseteq_\gamma \gamma'}$$

$$(14)\quad \frac{\forall EAP \in \mathrm{dom}(S) : S(EAP) \sqsubseteq_V S'(EAP)}{S \sqsubseteq_S S'}$$

$$(15)\quad \frac{\forall (B', V') \in q' : \exists (B, V) \in q : B' \sqsubseteq_B B \wedge V \sqsubseteq V'}{q \sqsubseteq_q q'}$$

$$(16)\quad \frac{\forall (B', S') \in s' : \exists (B, S) \in s : B' \sqsubseteq_B B \wedge S \sqsubseteq_S S'}{s \sqsubseteq_s s'}$$

$$(17)\quad \frac{\forall B \in b : \exists B' \in b' : B \sqsubseteq_B B'}{b \sqsubseteq_b b'}$$

그림 6 분석 의미공간의 순서 정의

가 포섭한다는 것이다. 다시 말해, h'가 h보다 더 요약된 메모리 업데이트들의 나열이라는 것이다. 예를 들면, 그림 6의 규칙 (8)은 만약 AP와 AP'이 같은 메모리 위치를 나타내지 않는 경우에는, 그 두 업데이트 순서를 바꾸어도 상관이 없다는 것을 나타낸다. 규칙 (9)는, 만약 AP와 AP'가 같은 메모리 위치를 나타내는지 나타내지 않는지 현재의 가정만으로는 판단할 수 없을 경우에는 AP의 업데이트가 AP'가 업데이트하는 값(V')을 함께 업데이트 해야만, AP를 AP'의 뒤로 위치를 바꾼 새로운 업데이트 나열이 원래의 업데이트 나열을 포섭한다는 것을 의미한다. 규칙 (6)은, 규칙 (8)과 (9)를 일반화 시켜, 임의의 업데이트를 임의의 횟수만큼 뒤로 옮길 수 있도록 하는 규칙이다. 참고로, 업데이트 기록처럼 수학적인 의미에서의 나열(sequence)은 그 순서의 정의가 프로그램의 의미와 잘 연계되지 않아 프로그램 분석 공간으로 잘 활용되지 않았는데, 위와 같이 업데이트 기록의 순서를 프로그램의 실행 의미와 연계해 안전하게 재구성 하는 방법을 고안함으로써 업데이트 기록간의 순서를 정의 할 수 있었고, 이에 따라 메모리 반응의 순서에 대한 정보를 유지하는 분석 공간을 정형화 할 수 있었던 점도 본 연구의 중요한 결과이다. 결과적으로 업데이트 기록은 1장에서 요약하였고 차후 3.3절, 3.4절에서 자세히 설명하는 바와 같이 분석기가 메모리 갱신 명령(예, 할당문, 함수 호출)이나 읽기 명령(예, 표현식)을 만났을 때 강한 업데이트가 자연스럽게 수행 할 수 있도록 해주고, 관련 별칭 문맥을 효율적으로 찾는 데에도 도움을 준다.

우리의 기본 분석은 if($s1$, $s2$)에서 s_1의 업데이트 기록과 s_2의 업데이트 기록을 집합(*Histories*)으로 모으는 형태로 정형화되었다. 따라서, 경로에 따른 실행의 차이를 추적 할 수 있다. 하지만 이로 인해 분석의 성능이 떨어질 수 있는데, 이에 대한 해결 방법은 차후 3.6 절에서 자세히 다룰 것이다.

실행요약 $\{(B_1, H_1), (B_2, H_2), ...\} \in AbstractStatement$

의 의미는 함수 호출 시점에서 B_1이 만족된다면, 함수의 실행 결과는 H_1으로 안전하게 요약된다는 것이고, 마찬가지로 B_2가 만족되었다면, 요약 결과는 H_2라는 것이다. 또한, 함수 호출 시 입력 메모리가 2개 이상의 B_i 들을 만족시킬 경우는 어느 결과를 취해도 상관이 없다는 것을 의미한다. 이 의미공간 또한 다른 의미공간과 마찬가지로 논리적 암시를 염두에 두고 순서($\sqsubseteq_d$)를 정의하였다. 이 순서의 직관적인 의미는 다음과 같이 두 가지로 나누어 생각할 수 있다: (1) $d \in$ *AbstractStatement*에서 임의대로 원소를 제거하여도 (즉, $d' \sqsubseteq d$인 d') 안전하다. 왜냐하면 $d = \{(B_1, H_1), ..., (B_i, H_i)\}$의 의미는 B_1이 만족되면 H_1이라 할 수 있고, ..., B_i이 만족되면 H_i라 할 수 있다는 것이기 때문에, 우리는 B_1이 만족되면 H_1이라 할 수 있고, ..., B_{i-1}이 만족되면 H_{i-1}이라고 할 수 있다. 즉 $d' = \{(B_1, H_1), ..., (B_{i-1}, H_{i-1})\}$이라 하면, d가 d'를 논리적으로 암시한다고 볼 수 있다. (2) 만약 d'에 d에 존재하지 않는 (B', H')가 들어있다면, 어떠한 (B, H)가 d에 존재하여서, B'인 가정 하에서 해당 항목((B, H))을 사용할 수 있고(즉 B'가 B를 논리적으로 암시, $B' \sqsubseteq_B B$), B' 가정 하에서 H'이 H를 포섭하는 경우에만 d'이 d을 포섭한다고 말할 수 있다.

시그너쳐(*Signatures*)는 함수이름에서 해당 함수의 매개변수들로의 매핑이고, 요약 환경 ($\widehat{Env}$)은 각 함수에 대한 실행요약 모음이다. 1장에서 설명했듯이 우리는 함수의 호출이 있을 때 그 함수의 몸통을 보지 않고 요약 환경에 기록해둔 실행요약을 이용하여 함수 호출을 분석한다. 치환(*Substitution*)은 함수가 호출될 때, 호출된 함수의 매개변수의 관점에서 표현된 각 확장 접근 경로들을 호출하는 함수의 관점에서 표현된 접근 경로들로 매핑 시켜주는 함수이다. $2^{Assumptions \times \widehat{Value}}$ 및 $2^{Assumptions \times Substitution}$은 실행요약과 비슷하게 설명될 수 있다. 즉, $\{(B_1, V_1), ..., (B_n, V_n)\} \in 2^{Assumptions \times \widehat{Value}}$의 의미는, B_i가 만족된다면 $\widehat{Value}$가 V_i임을 뜻한다. 마지막으로 $2^{Assumptions}$는 가정들의 모임이다. $\{B_1, ..., B_n\}$의 의미는 B_1 또는 B_2 또는 ... B_n이 만족된다는 것을 의미한다. 포함집합관계($\supseteq$) 대신에, 이미 가정들 사이에도 순서($\sqsubseteq_B$)가 정의되어 있으므로 이것을 이용하여 $b \sqsubseteq_b b'$을 b가 b'를 논리적으로 암시한다는 것을 의미하도록 정의하였다.

참고로, 본 절에서 정의된 의미공간들이 모두 분석의 기본 의미 공간인 파샬오더(partial order)가 되는 것은 아니며, 일부 의미공간은 프리오더(perorder)임에 주지하자. 예를 들어, $B = \{AP\#AP'\}$와 $B' = \{AP'\#AP\}$을 살펴보자. $\sqsubseteq_B$의 정의에 의하면, $B \sqsubseteq_B B'$이고 $B' \sqsubseteq_B B$이지만, 명확히 $B \neq B'$이다. 하지만, 임의의 프리오더는 다음과 같은 동등 조건(equivalence)를 사용하여 프로그램 분석에서 사용될 수 있는 파샬오더로 만들 수 있다[26].

$$X = X' \text{ if and only if } X \sqsubseteq X' \wedge X' \sqsubseteq X$$

위의 동등 조건의 정의를 취하면, 그림 5의 의미공간들은 모두 래티스(lattice) 혹은 세미-래티스(semi-lattice)가 된다.

3.3 업데이트 기록 읽기 및 쓰기 동작

본 절에서는 3.2절에서 정의한 우리의 핵심 분석 공간인 업데이트 기록이 분석 중 어떻게 사용 되는지 자세

Assumptions	$\top_B = \{\}$ $B \sqcap_B B' = B \cup B'$
$\widehat{Value}$	$\bot_V = \{\}$ $V \sqcup_V V' = V \cup V'$
AbstractStatement	$\bot_d = \{(\top_B, \{\})\}$ $\top_d = \{\}$ $d \sqcup_d d' = \{(B \sqcap_B B', H \cup H') \mid (B, H) \in d, (B', H') \in d'\}$ $d \sqcap_d d' = d \cup d'$
$\widehat{Env}$	$\bot_\gamma = \lambda f.\bot_d$ $\gamma \sqcup_\gamma \gamma' = \lambda f.\gamma(f) \sqcup_d \gamma'(f)$
Substitution	$\bot_S = \lambda x.\bot_V$ $S \sqcup_S S' = \lambda x.S(x) \sqcup_V S'(x)$
$2^{Assumptions \times \widehat{Value}}$	$\bot_q = \{(\top_B, \bot_V)\}$ $\top_q = \{\}$ $q \sqcup_q q' = \{(B \sqcap_B B', V \sqcup_V V') \mid (B, V) \in q, (B', V') \in q'\}$ $q \sqcap_q q' = q \cup q'$
$2^{Assumptions \times Substitution}$	$\bot_s = \{(\top_B, \bot_S)\}$ $\top_s = \{\}$ $s \sqcup_s s' = \{(B \sqcap_B B', S \sqcup S') \mid (B, S) \in s, (B', S') \in s'\}$ $s \sqcap_s s' = s \cup s'$
$2^{Assumptions}$	$\bot_b = \{\}$ $\top_b = \{\top_B\}$ $b \sqcup_b b' = b \cup b'$ $b \sqcap_b b' = \{B \sqcap_B B' \mid B \in b, B' \in b'\}$

그림 7 분석 의미공간의 유추된 연산

히 기술 한다.

먼저 프로그램식을 계산할 때 주어진 메모리(업데이트 기록)을 읽기 위해 사용되는 read(B, h, V)를 살펴보자. read(B, h, V)는 호출 문맥에 대한 가정이 B이고 현재까지의 메모리 업데이트가 h라고 할 때, V가 나타내는 AP들이 가지는 값(즉 메모리 위치 AP가 나타내는 주소가 가리키는 값)을 계산한다. read(B, h, V)는 $\text{read}_{AP}(B, h, AP)$를 이용하여 V에 있는 각각의 AP에 대해 그 값을 계산한 다음, 그 값들을 모두 포섭하기 위해 $\sqsubseteq_q$로 합친다.

read_{AP} (B, h, AP)는 필요하면 주어진 AP가 h에 나타나있는 다른 접근경로들에 별칭관계를 B를 이용하여 체크하는데, 주어진 B 하에서 결론이 나오지 않을 경우, 주어진 B에 새로운 가정을 추가한 새로운 B'에 대해서도 결과를 구해준다. 즉, read_{AP} (B, h, AP)의 결과는 $\{(B_1, V_1), ..., (B_n, V_n)\}$의 형태가 된다.

read_{AP}의 동작은 크게 주어진 업데이트 기록이 ε일 때와 그렇지 않을 때로 경우가 나뉜다.

업데이트 기록 ε은 업데이트가 일어나지 않은 상태, 즉 함수의 초기 입력 메모리를 뜻하므로, 함수가 호출될 시점에서 AP의 (AP가 가리키는)값 $AP*$ 가 read_{AP}의 결과가 된다. 하지만, 주어진 AP는 이미 접근 경로 요약함수 α에 의해 요약이 된 상태인 반면 (알고리즘에서는 항상 요약된 접근 경로만을 사용한다), $AP*$는 임의로 만들어 낸 접근 경로이기 때문에, $\alpha(AP*)$와 같이 요약을 해야 한다.

이번에는 업데이트 기록이 $h{\cdot}[AP' \rightarrow V]$인 경우를 생각해 보자. 먼저 그림 8에 정의된 collapsed 함수의 의미를 살펴보자. 어떤 AP가 뭉쳐졌음(collapsed)은 추상화 시켰을 때 AP가 되는 두 개 이상의 접근 경로가 있다는 것이다. 즉, AP는 실제로는 두 개 이상의 메모리 주소를 하나로 이름으로 지칭하고 있음을 뜻한다. 반대로 뭉쳐지지 않았다고 하면, 실행 시 단 하나의 메모리 주소를 나타낸다. 이제 read_{AP}로 되돌아가자. AP와 AP'이 서로 같고, AP가 뭉쳐지지 않았다면, 확실히 $[AP' \rightarrow V]$가 AP에 대한 마지막 업데이트라고 할 수 있다. 따라서 결과는 V가 된다. AP와 AP'가 서로 같지만, AP'이 뭉쳐진 접근 경로라면 이를 AP에 대한 마지막 업데이트라고 할 수 없다. 예들 들어 다음과 같은 경우를 생각해보자. ap와 ap'는 서로 다르고 둘 다 추상화시키면 AP가 된다고 해보자. 이 상황에서는 $[AP' \rightarrow V']$가 ap에 대한 기록인지 ap'에 대한 기록 인지 구분할 수가 없다. 만약 ap'에 의해서 생긴 업데이트 기록이었다고 해보자. 또한 현재 AP를 read하려는 것이 ap를 read해야 하기 때문인지 ap'를 read해야 하기 때문인지 구분할 수 없다. 실제로는 ap의 값을 알아내기 위한 것이었다고 해보자. 이러한 경우에는 ap의 실제 값은 V'가 아닌 (왜냐하면 ap랑 ap'가 실제로 다른 메모리 위치였기 때문에), 함수호출 시점에서 ap위치에 들어있던 값 그대로가 된다. 반면에 ap'의 값을 알아내기 위한 read 였다면, 이번에는 결과가 V'이 된다. 실제로 어느 경우인지는 알 수 없으므로 (접근 경로의 추상화 때문에 잃은

$$
\begin{array}{l}
\text{read}(B,h,V) = \sqcup_q\{\text{read}_{AP}(B,h,AP) \mid AP \in V\} \\
\\
\text{read}_{AP}(B,\epsilon,AP) = \{(B,\{\alpha(AP*)\})\} \\
\text{read}_{AP}(B,h\cdot[AP' \mapsto V],AP) = \\
\quad \texttt{if } AP = AP' \wedge \neg\text{collapsed}(AP) \texttt{ then} \\
\qquad \{(B,V)\} \\
\quad \texttt{else if } AP = AP' \wedge \text{colllapsed}(AP) \texttt{ then} \\
\qquad \{(B',V' \sqcup_V V) \mid (B',V') \in \text{read}_{AP}(B,h,AP)\} \\
\quad \texttt{else if } B \vdash AP\#AP' \texttt{ then} \\
\qquad \text{read}_{AP}(B,h,AP) \\
\quad \texttt{else} \\
\qquad \texttt{let } q = \text{read}_{AP}(B,h,AP) \texttt{ in} \\
\qquad \{(B',V' \sqcup_V V) \mid (B',V') \in q\} \sqcap_q \{(B' \sqcap_B \{AP\#AP'\},V') \mid (B',V') \in q\} \\
\\
\text{update}(B,h,V_1,V_2) = \{\text{update}_{AP}(B,h,AP,V_2) \mid AP \in V_1\} \\
\\
\text{update}_{AP}(B,h,AP,V) = \\
\quad \texttt{if } \neg\text{collapsed}(AP) \texttt{ then } \text{remove}(h,AP)\cdot[AP \mapsto V] \\
\quad \texttt{else } \text{remove}(h,AP)\cdot[AP \mapsto V \sqcup_V \text{fetch}(h,AP)] \\
\\
\text{collapsed}(y*^m) = m \geq \kappa(y) \quad (\text{i.e., } \exists ap : ap \neq y*^m \wedge \alpha(ap) = y*^m) \\
\\
\text{remove}(\epsilon,AP) = \epsilon \\
\text{remove}(h\cdot[AP' \mapsto V],AP) = \texttt{if } AP = AP' \texttt{ then } h \texttt{ else } \text{remove}(h,AP)\cdot[AP' \mapsto V] \\
\\
\text{fetch}(\epsilon,AP) = \bot_V \\
\text{fetch}(h\cdot[AP' \mapsto V],AP) = \texttt{if } AP = AP' \texttt{ then } V \texttt{ else } \text{fetch}(h,AP)
\end{array}
$$

그림 8 업데이트 기록 읽기 및 쓰기

정보라고 할 수 있다), 우리는 안전하게 이전의 업데이트 기록인 h에서 AP를 읽은 값인 V와, V'을 합해준다. 이번에는 주어진 B하에서 AP와 AP'의 별칭이 아니라고 가정된 경우를 생각해 보자. 이는 곧 실행 시 다른 메모리 주소임을 보장하므로, AP'에 대한 업데이트는 AP에는 아무런 영향을 끼치지 않는다. 따라서 앞의 업데이트 기록인 h에서 AP의 값을 알아내면 된다. 위의 세가지 경우가 모두 아닌 경우는 바로 AP와 AP'가 서로 다르면서, 주어진 B하에서 AP가 AP'의 별칭이 아니라고 결론을 내릴 수 없는 (모르는) 경우이다. 만약 AP가 AP'의 별칭이라면 $[AP' \rightarrow V']$이 AP에 대한 마지막 업데이트인지 아닌지 확실히 알 수가 없기 때문에, 이전업데이트 기록 하에서의 AP의 값과 V'을 합쳐야 한다. 그렇지 않고 AP가 AP'의 별칭이 아니라면, h에서만 AP를 읽으면 된다. 즉, AP와 AP'가 별칭관계가 있는지 여부에 따라 결과가 다르게 나오므로, 이때 우리는 주어진 가정 B에 $AP\#AP'$(즉 AP와 AP'은 다르다)라는 가정을 추가한 경우와 B 그대로인 경우 둘 다에 대해서 계산해 준다. 이런 식으로 우리의 분석은 함수의 실행을 달라지게 하는 관련 별칭 문맥들을 필요하면 분석 중 도입하게 됨에 주지하자. 또한, 이러한 이유로 $2^{Assumptions} \times \widehat{Value}$ 의미공간이 사용된다.

이제 명령문을 분석할 때 메모리 상태 갱신을 위해 사용되는 update(B, h, V_1, V_2)를 살펴 보자.

update(B, h, V_1, V_2)은 V_1이 나타내는 각 AP 들에 대해 V_2로 업데이트 하는 명령을 h에 추가시킨다. 이때, V_1이 나타내는 AP들이 여러 개 있을 경우는, 실제로는 어떤 AP가 업데이트 되어야 하는지 알 수 없으므로, 각 AP들이 업데이트되는 경우를 모두 포섭하도록 해야 한다. 따라서 AP에 대해 업데이트 명령을 추가시키는 $\text{update}_{AP}(B, h, AP, V_2)$를 통해 얻어진 $History$들을 모두 포섭하여 $Histories$로 만든다.

update_{AP} (B, h, AP, V)는 주어진 $History$인 h에 주어진 AP를 V로 갱신한 결과 업데이트 기록을 계산해 낸다. 이것의 동작은 두 가지 경우로 나뉜다. 주어진 AP가 뭉쳐진 접근 경로가 아니라면, h에서 AP에 대한 기존의 업데이트를 제거시킨 후, 새로운 업데이트 기록 $[AP \rightarrow V]$을 제거된 h 뒤에 추가시킨다. 왜냐하면, 먼저 read에서 설명되었듯이, 현재 추가하는 업데이트 명령이 AP에 대한 최신의 업데이트라고 할 수 있기 때문에, 이 시점부터는 기존의 AP에 대한 업데이트 명령은 안전하게 무시할 수 있기 때문이다. 만약 AP가 뭉쳐진 접근 경로라면, 기존의 업데이트 명령을 그냥 없애버려서는 안 된다. 앞서 read에서 설명 되었듯이, AP는 두 개 이상의 접근 경로를 의미하므로, 기존의 업데이트와 현재 추가하는 업데이트가 실제로 나타내는 대상이 서로 다를 수가 있기 때문이다. 따라서 기존의 업데이트 기록에서 AP와 관련될 수 있는 갱신 값을 fetch 함수를 이용 해 찾아낸 후, 기존의 AP에 대한 업데이트를 제거시키고, 현재 추가하려는 값과 뭉쳐서 ($\sqcup_V$) AP에 대한 하나의 업데이트 기록을 남긴다. 이런 식으로 $History$의 업데이트 대상이 되는 AP들은 단 한번씩만 나타날 수 있기 되기 때문에, $History$의 최대길이는 $|\widehat{AccessPath}|$로 제한됨에 주지하자.

3.4 분석 알고리즘

그림 9는 프로그램 p의 각 함수의 실행요약($Abstract\text{-}Statement$)을 구해내는 우리의 주 알고리즘이다.

$I_e(e, B, h)$는 B의 조건들이 만족된다고 가정한 상황에서, 그리고 지금까지 수행된 메모리 업데이트들이 h라고 가정할 때, e가 나타내는 값을 알아내기 위해 사용된다. 하지만 e의 값을 계산하는 도중에, 현재 주어진 B만으로는 부정확한 결과가 나올 수 있다. 따라서 $I_e(e, B, h)$는 현재 주어진 B에 무엇인가를 추가로 더 가정할 경우, 더 자세한 결과를 얻을 수 있을 경우에, 주어진 B에 해당 가정을 추가한 상황에 대해서도 e의 값을 계산해 준다. 다시 말해 $I_e(e, B, h)$는 $\{(B_1, V_1), ..., (B_n, V_n)\}$ 형태의 결과를 돌려주는데, 이것은 만약 B_i가 만족된다면, e의 값을 V_i라고 할 수 있다는 것을 의미한다.

$I_e(\&x, B, h)$는 메모리 위치 x를 나타내는 주소값을 반환한다. 즉, $\{x\}$가 된다. $I_e(*e, B, h)$에서는 e를 계산하여 나온 접근경로들이 가리키는 접근경로들을 read 함수를 사용하여 계산한다. 이때 3.3절에서 설명된 것처럼 e의 값을 계산 할 때 관련 별칭 문맥들을 필요에 따라 도입하고, read시에도 마찬가지로 관련 별칭 문맥들을 필요에 따라 도입한다. $I_e(null, B, h)$는 아무 곳도 가리키지 않는 값, 즉, $\perp_V$를 반환한다.

$I_s(s, \beta, \gamma, d)$는 함수들의 실행요약 환경 γ하에서 명령문 s에 도달하기 전까지 나타났던 명령문들의 실행요약이 d라고 할 때, s의 메모리 반응을 d에 추가시킨 새로운 실행요약을 계산한다. 주어지는 d에는 (B, H)들이 들어 있는데, 이것의 의미는, 함수호출 시점에서 B가 만족된다면, 지금까지 만난 명령문들의 실행을 H라 요약할 수 있다는 뜻이다. H 내에서 각 $h \in History$는 s에 도달하기까지 가능했던 모든 메모리 반응들을 요약한다.

$I_s(*e_1 := e_2, \beta, \gamma, d)$는 지금까지의 실행요약 d에 있는 각 (B, H)의 각 $h \in H$에 대해, e_1이 나타내는 값(주소)과 e_2이 나타내는 값을 계산한다. 그런 후, h에 e_1이 나타내는 메모리 위치들에 e_2의 값을 3.3절에서 설명한 update(B, h, V', V'') 함수를 통해 계산한다. 이때, e_1이 나타내는 값 V'는 가정 B'하에서, 그리고 e_2가 나타내는 값 V''는 가정 B''하에서, 계산된 것이므로, $B' \sqcap_B B''$ 하에서 e_1, e_2 각각 V', V'' 값을 갖는다고 할 수 있

$$
\begin{array}{l}
I_e : Expression \times Assumptions \times History \rightarrow 2^{Assumptions \times \widehat{Value}} \\
I_e(\&x, B, h) = \{(B, \{x\})\} \\
I_e(*e, B, h) = \texttt{let } q = I_e(e, B, h) \texttt{ in } \sqcap_q\{\text{read}(B', h, V') \mid (B', V') \in q\} \\
I_e(\text{null}, B, h) = \{(B, \bot_V)\} \\
\\
I_s : Statement \times Signatures \times \widehat{Env} \times AbstractStatement \rightarrow AbstractStatement \\
I_s(*e_1 := e_2, \beta, \gamma, d) = \\
\quad \sqcap_d\{\sqcup_d\{\sqcap_d\{ \\
\quad \{(B' \sqcap_B B'', \text{update}(B, h, V', V''))\} \mid (B', V') \in I_e(e_1, B, h), (B'', V'') \in I_e(e_2, B, h) \\
\quad \} \mid h \in H\} \mid (B, H) \in d\} \\
I_s(f(\overline{e_i}), \beta, \gamma, d) = \\
\quad \texttt{let } (\overline{y_i}, d') = (\beta(f), \gamma(f)) \texttt{in} \\
\quad \sqcap_d\{\sqcup_d\{\sqcap_d\{ \\
\quad \text{apply}(B', h, d', S) \mid (B', S) \in \text{make-subst}(h, \{\overline{y_i \mapsto I_e(e_i, B, h)}\}, \text{eaps-of}(d')) \\
\quad \} \mid h \in H\} \mid (B, H) \in d\} \\
I_s(s_1; s_2, \beta, \gamma, d) = I_s(s_2, \beta, \gamma, I_s(s_1, \beta, \gamma, d)) \\
I_s(\texttt{ if } (s_1, s_2), \beta, \gamma, d) = I_s(s_1, \beta, \gamma, d) \sqcup_d I_s(s_2, \beta, \gamma, d) \\
I_s(\texttt{ skip }, \beta, \gamma, d) = d \\
\\
I_{cl} : Closure \times Signatures \times \widehat{Env} \rightarrow AbstractStatement \\
I_{cl}((\overline{y_i}, \overline{z_j}, s), \beta, \gamma) = (\overline{y_i}, I_s(s, \beta, \gamma, \{(\top_B, \{\epsilon \cdot \overline{[z_j \mapsto \{\}]}\})\}) \backslash \{\overline{y_i}, \overline{z_j}\}) \\
\\
I_p : Program \rightarrow \widehat{Env} \\
I_p(p) = \\
\quad \texttt{let } \beta = \lambda f.\overline{y_i} \text{ where } (\overline{y_i}, \_, \_) = p(f) \\
\quad \texttt{let } F(\gamma) = \lambda f \in \text{dom}(p).I_{cl}(p(f), \beta, \gamma) \texttt{ in} \\
\quad \texttt{let } G(\gamma) = \texttt{if } F(\gamma) \sqsubseteq_\gamma \gamma \texttt{ then } \gamma \texttt{ else } G(F(\gamma)) \texttt{ in} \\
\quad G(\bot_\gamma)
\end{array}
$$

그림 9 분석 알고리즘

다. 따라서 그 결과는 ($B' \sqcap_B B''$, update(B, h, V', V'')) 이 된다. 각 (B', V') 및 (B'', V'')로부터 얻은 결과들끼리는 서로 대등하므로 $\sqcap_d$를 이용하여 결과를 모은다. 그 후, 각 $h \in H$에 대한 결과를 모두 포섭해야 하므로 $\sqcap_d$를 이용하여 합친다. 마지막으로 d에 있는 각 (B, H)에 대한 결과들은 서로 대등하므로, $\sqcap_d$로 합친다.

$I_s(f(\overline{e_i}), \beta, \gamma, d)$는 현재까지의 실행요약 d에 함수 f의 실행요약을 이용하여 메모리 반응을 추가시킨 새로운 실행요약을 계산한다. 먼저 넘겨받은 β와 γ에서 호출하려는 함수 f의 $\overline{y_i} \in FormalVar^*$ 및 $d' \in AbstractStatement$를 꺼내온다. 그리고, d에 있는 각각의 (B, H)에 대해 이제 함수 f의 요약결과 d'를 적용한다. 이 과정은 H에 있는 각각의 h에 대해 따로 이루어진다. $h \in H$에 대해, 함수 f의 매개변수로부터 만들어지는 확장 접근 경로들로부터 현재 호출하는 함수의 상황에서의 실제 메모리 위치들로의 매핑 S를 만든다. 그림 10에는 함수호출 처리를 위해 사용되는 분석함수들이 정의되어 있다. eaps-of(d)는 d안에 나타나는 모든 확장 접근 경로들을 나타낸다. make-subst(h, $\{\overline{y_i \mapsto I_e(e_i, B, h)}\}$, eaps-of($d'$))는 eaps-of($d'$)의 각각의 EAP가 가정 B와 h에서 어떤 주어진 $EAPs$에서부터 그 $EAPs$들이 현재 h에서 나타내는 메모리 위치들로의 매핑을 만든다. 이 때, 다른 분석함수들과 마찬가지로 필요하면 (가정이 불충분한 경우), 더 자세한 새로운 가정들을 만들어가면서 매핑을 만들어나간다. d'에 나타나는 가정들이나 업테이트 기록들은 모두 함수 f의 매개변수로부터 시작되는 접근 경로들로 나타나 있기 때문에, h와 S를 이용하여 현재 문맥상의 AP들로 치환시키는 과정이 필요하다. apply(B, h, d', S)는 d'에 있는 각 (B_i, H_j)에 대해, B_i가 현재 호출문맥에서 만족되는지 refine(B, S, B_i)를 이용하여 체크한다. 만약 현재 가정 B으로 충분치 않다면, 새로운 가정들이 추가된 새로운 B'이 refine에 의해 계산된다. 그런 후에, h에 H_j에 들어있는 각 h_k를 append 함수를 이용하여 덧붙인다. 이 모든 결과를 포섭해야 하므로 $\cup$로 결과를 합쳐 H'를 얻는다. 최종적으로 (B_i, H_j) $\in d$에 대한 결과는 (B', H')가 된다.

$I_s(s_1; s_2, \beta, \gamma, d)$는 d에 s_1의 메모리 반응과 s_2의 메모리 반응을 연이어 계산한다.

$I_s(\text{if}(s_1, s_2), \beta, \gamma, d)$는 d에서 s_1을 수행한 실행요약과 s_2을 수행한 실행요약을 $\sqcup_d$를 이용하여 합친다.

$I_{cl}((\overline{y_i}, \overline{z_j}, s), \beta, \gamma)$는 프로그램 상에 나타나는 모든 함수들의 지금까지 계산된 실행요약들($\gamma \in \widehat{Env}$)에 기반하여, 주어진 함수의 실행요약을 계산한다. 주어진 함수를 분석할 때 초기에 주어지는 실행요약은 $\{(\top_B, \{\epsilon \cdot \overline{[z_j \mapsto \{\}]}\})\}$ 이다. 즉, 지역변수들을 초기값을 최소값 (bottom, {})으로 설정하고 분석한다. 따라서 함수의 몸통 s를 분석하는 도중, 지역변수를 초기화하지 않고 read하게 되면, {}이 나오게 된다. 요약이 끝난 후, 매개변수 및 지역변수

$$
\begin{array}{l}
\mathrm{apply}(B,h,d,S) = \{(B',H') \mid (B_i,H_j) \in d,\ B' \in \mathrm{refine}(B,S,B_i),\\
\qquad\qquad H' = \cup\{\mathrm{append}(B',h,S,h_k) \mid h_k \in H_j\}\}\\
\\
\mathrm{refine}(B,S,B_i) =\\
\quad \textbf{if } \exists (EAP\#EAP') \in B_i : S(EAP) \cap S(EAP') \neq \{\} \textbf{ then}\\
\quad\quad \{\}\\
\quad \textbf{else}\\
\quad\quad \{B \sqcap_B \{AP\#AP' \mid (EAP\#EAP') \in B_i, AP \in S(EAP), AP' \in S(EAP')\}\}\\
\\
\mathrm{append}(B,h,S,\epsilon) = \{h\}\\
\mathrm{append}(B,h,S,h' \cdot [EAP \mapsto V]) =\\
\quad \cup\{\mathrm{update}(B,h'',S(EAP),\sqcup_V\{S(EAP') \mid EAP' \in V\}) \mid h'' \in \mathrm{append}(B,h,S,h')\}\\
\\
\mathrm{make\text{-}subst}(h,T,EAPs) =\\
\quad \sqcup_s\{\{(B',\{EAP \mapsto V'\}) \mid (B',V') \in \mathrm{substitute}(h,T,EAP)\} \mid EAP \in EAPs\}\\
\\
\mathrm{substitute}(h,T,y{*}^{m+1}) =\\
\quad \sqcup_q\{\sqcap_q\{\mathrm{iter\text{-}read}(B,h,V,n) \mid (B,V) \in T(y)\} \mid \alpha(y{*}^n) = y{*}^m\} =\\
\quad \textbf{let } \ q_1 = \sqcap_q\{\mathrm{iter\text{-}read}(B,h,V,m) \mid (B,V) \in T(y)\} \textbf{ in}\\
\quad\quad \textbf{if } m < \kappa(y) \textbf{ then}\\
\quad\quad\quad q_1\\
\quad\quad \textbf{else}\\
\quad\quad\quad \mathrm{lfp}\ \lambda q.(\{\sqcap_q\{\mathrm{iter\text{-}read}(B,h,V,\omega(y)) \mid (B,V) \in q\}\} \sqcup_q q_1)\\
\\
\mathrm{iter\text{-}read}(B,h,V,0) = \{(B,V)\}\\
\mathrm{iter\text{-}read}(B,h,V,n+1) = \sqcap_q\{\mathrm{read}(B',h,V') \mid (B',V') \in \mathrm{iter\text{-}read}(B,h,V,n)\}
\end{array}
$$

그림 10 함수호출관련 함수들

들은 스택 변수이므로 요약결과에서 모두 제거한다. 즉 그 결과에서 $\widehat{ExtendedAccessPath}$만을 남겨둔다.

$I_p(p)$는 주어진 프로그램 p로부터 그 프로그램 상의 모든 함수들의 실행요약들을 계산한다. 초기 $\widehat{Env}$는 $\bot_\gamma$으로 모든 함수들의 실행요약이 $\{(\top_B, \{\})\}$이다. 이것의 의미는, 항상 종료되지 않는 실행요약이다. $\bot_\gamma$에서 시작하여, 고정점을 계산해 낸다. 3.2절에서 설명한 바와 같이 우리의 분석 의미공간들은 유한한 세미-래티스(semi-lattice) 혹은 래티스(lattice)이며, 분석 함수들은 단조증가함수(monotonic function)들이기 때문에, 알고리즘은 항상 종료된다.

3.5 분석 예제

우리의 언어로 쓰여진 다음 함수의 실행요약(*Abstract-Statement*)을 계산해 보자.

```
f((y1, y2, y3, y4), ()) =
    *y3 := y4;
    if(*y1 := *y2, *y4 := *y2)
```

이때 분석에서 각 y_i에 대해 $eap(y_i{*}) = y_i{*}$이고 $eap(y_i{*}{*}) = y_i{*}{*}$가 되는 접근 경로 요약 함수를 사용한다고 가정하자. 3.4절에서 설명된 바와 같이 함수의 시작점에서 실행요약은 $\{(\top_B, \{\varepsilon\})\}$이고, 첫 번째 명령문 $*y3 := y4$를 지난 후의 실행요약은 $\{(\top_B, \{\varepsilon \cdot [y3 \rightarrow \{y4{*}\}]\})\}$이 된다. 이 실행요약을 가지고 if문의 then부분의 명령문을 처리하면 다음의 결과를 얻는다.

$\{\ (\top_B, \{\varepsilon \cdot [y3 \rightarrow \{y4{*}\}][y1{*} \rightarrow \{\ y2{*}{*},\ y4{*}\}]\}),$

$(\{y2{*}\#y3{*}\}, \{\varepsilon \cdot [y3{*} \rightarrow \{y4{*}\}][y1{*} \rightarrow \{y2{*}{*}\}]\})\ \}$

이때, 새로운 가정이 생긴 이유는, $y2{*}$을 업데이트 기록에서 읽는 과정에서 $y3{*}$과 별칭 관계인지 여부에 따라 실행 결과가 달라지기 때문이다. 이번에는 $\{(\top_B, \{\varepsilon \cdot [y3 \rightarrow \{y4{*}\}]\})\}$을 가지고 if문의 else부분의 명령문을 처리하면 다음의 결과를 얻는다.

$\{\ (\top_B, \{\varepsilon \cdot [y3{*} \rightarrow \{y4{*}\}][y4{*} \rightarrow \{y2{*}{*},\ y4{*}\}]\}),$

$(\{y2{*}\#y3{*}\}, \{\varepsilon \cdot [y3{*} \rightarrow \{y4{*}\}][y4{*} \rightarrow \{y2{*}{*}\}]\})\ \}$

이제 두 실행요약을 모두 포섭하게끔 $\sqcup_d$로 합치면 다음과 같이 된다.

$\{\ (\top_B, \{\varepsilon \cdot [y3{*} \rightarrow \{y4{*}\}][y1{*} \rightarrow \{y2{*}{*},\ y4{*}\}],$

$\varepsilon \cdot [y3{*} \rightarrow \{y4{*}\}][y4{*} \rightarrow \{y2{*}{*},\ y4{*}\}]\}),$

$(\{y2{*}\#y3{*}\}, \{\varepsilon \cdot [y3{*} \rightarrow \{y4{*}\}][y1{*} \rightarrow \{y2{*}{*},\ y4{*}\}],$

$\varepsilon \cdot [y3{*} \rightarrow \{y4{*}\}][y4{*} \rightarrow \{y2{*}{*}\}]\}),$

$(\{y2{*}\#y3{*}\}, \{\varepsilon \cdot [y3{*} \rightarrow \{y4{*}\}][y1{*} \rightarrow \{y2{*}{*}\}],$

$\varepsilon \cdot [y3{*} \rightarrow \{y4{*}\}][y4{*} \rightarrow \{y2{*}{*},\ y4{*}\}]\}),$

$(\{y2{*}\#y3{*}\}, \{\varepsilon \cdot [y3{*} \rightarrow \{y4{*}\}][y1{*} \rightarrow \{y2{*}{*}\}],$

$\varepsilon \cdot [y3{*} \rightarrow \{y4{*}\}][y4{*} \rightarrow \{\ y2{*}{*}\ \}]\})\ \}$

3.6 알고리즘의 개선

앞 예제의 최종결과를 보면 $\{y2{*}\#y3{*}\}$에 대한 결과가 3개가 있다. 이 3개의 결과 모두 $\{y2{*}\#y3{*}\}$가 만족되는 상황에서는 옳은 실행요약이다. 이 3개 사이의 정확도(순서, $\sqsubseteq_d$)를 비교해 보면, 마지막 결과가 나머지 2개보다 더 정확함을 알 수 있다. 이것은 곧 나머지 2개의 결과는 사실상 쓸모없음을 암시하므로, 이 2개의 결과를

제거해 버릴 수 있다면, 알고리즘이 더 이상 쓸모 없는 항목들을 처리하지 않아도 되므로 더 효율적이 될 것이다. 또한 앞 예제에서는 드러나지 않지만, 새로운 가정들을 도입하는 것이 전혀 도움이 되지 않는 경우가 생긴다. 이런 경우 또한 해당 가정에 따라 계산한 결과를 제거할 수 있다면 좋을 것이다. 또한 분석하는데 걸리는 시간을 줄이기 위해, 도입될 수 있는 가정의 개수를 제한시킬 수 있다면 좋을 것이다.

3.5절의 알고리즘의 또 다른 개선점으로는 여러 업데이트 기록들을 모두 포섭하는데 사용하는 $\cup$을 들 수 있다. 분석함수들의 많은 부분에서 입력으로 들어온 $H \in Histories$보다 원소의 개수가 많은 새로운 업데이트 기록들을 생성하게 된다. 한 예로 update(B, h, V_1, V_2)를 들 수 있다. 이 경우, V_1에 들어있는 각 AP에 대해서 새로운 업데이트 기록이 만들어지며 그것들을 모두 집합으로 모으기 때문에 결국 $|V_1|$개의 업데이트 기록이 생성된다. 현재 알고리즘 상에서는 한번 생성된 업데이트 기록은 제거되지 않고 앞으로도 계속 처리가 되어야 하므로 분석하는데 오랜 시간이 걸릴 수 있다.

위의 두 가지 문제점, 즉 쓸모없는 결과가 생긴다는 점과 많은 수의 업데이트 기록이 생길 수 있다는 점은 둘 다 근본적으로 비슷한 방법으로 해결할 수 있다. 바로 팽창함수(extensive function)를 사용하는 것이다. 참고로 다음과 같은 성질을 만족하는 함수 w를 팽창함수라고 한다.

$$\forall x \in \mathrm{dom}(w) : x \sqsubseteq w(x)$$

f_1, $\cdots$, f_n가 모두 단조증가함수이고 w_0, w_1, $\cdots$, w_n이 팽창함수라고 해보자.

$f : X \to X = f_n \circ \cdots \circ f_1$ 이고 $g : X \to X = w_n \circ f_n \circ \cdots \circ w_1 \circ f_1 \circ w_0$라 하자. 그리고 X가 유한한 파샬오더라고 하자. 그러면 lfp $f \sqsubseteq$ pfp g이다. 여기서 lfp는 최소고정점 (least fixpoint)을, pfp는 임의의 앞고정점 (pre-fixpoint)을 의미한다. 그 이유는 다음과 같다.

$x \sqsubseteq w_0(x)$이다. $(f_k \circ \cdots \circ f_1)(x) \sqsubseteq (w_k \circ f_k \circ \cdots \circ w_1 \circ f_1 \circ w_0)(x)$라고 가정하자. 그러면, f_{k+1}가 단조함수이고 w_{k+1}은 팽창함수이므로 $(f_{k+1} \circ f_k \circ \cdots \circ f_1)(x) \sqsubseteq (f_{k+1} \circ w_k \circ f_k \circ \cdots \circ w_1 \circ f_1 \circ w_0)(x) \sqsubseteq (w_{k+1} \circ f_{k+1} \circ w_k \circ f_k \circ \cdots \circ w_1 \circ f_1 \circ w_0)(x)$이다. 따라서 $f(x) \sqsubseteq g(x)$이다. 이때, x가 g의 앞고정점이라고 해보자. 즉 $g(x) \sqsubseteq x$이라고 하자. 그러면 $f(x) \sqsubseteq x$이다. 즉, x는 f의 앞고정점이다. 그런데, lfp $f = \sqcup_{i \in w} f^i(\bot)$이다. $\bot \sqsubseteq x$로부터 임의의 i에 대해 $f^i(\bot) \sqsubseteq x$이므로 곧 lfp $f \sqsubseteq x$ 이다.

먼저 얻어진 결과에서 쓸모없는 결과를 없애는 방법에 대해 살펴보자. 실행요약의 순서($\sqsubseteq_d$) 정의에 따르면, $d = \{(B_1, H_1), \cdots, (B_n, H_n)\}$에서 임의의 개수의 (B_i, H_i)를 제거하면 원래의 d보다 더 커진다. 그림 11의 trim 함수는 이렇게 쓸모없는 결과를 제거하는 일을 한다. 실행 요약 뿐만 아니라, $2^{Assumptions \times \widehat{Value}}$와 같이 $2^{Assumptions \times X}$ 형태의 의미공간은 순서가 실행요약과 동등한 방법으로 정의되어 있으므로, 같은 방법으로 확장할 수 있다. 두 번째 문제점, 즉 분석 도중 업데이트 기록들이 너무 많이 생길 수 있는 문제점은, 분석 도중에 생성되는 모든 $H \in Histories$의 크기를 1이하로 제한시킴으로써 해결할 수 있다. 그림 11은 그 방법을 보여준다. 만약 $B \vdash h\ \nabla_h h' \Rightarrow h''$가 추론된다면, h''는 h와 h'둘 다 보다 더 큰 ($\sqsubseteq_h$에 의해) 업데이트 기록이다. 한편 $B \vdash \nabla_H H \Rightarrow \{h\}$가 추론된다면, h는 H의 각각의 업데이트 기록보다 더 큰 업데이트 기록이다. h와 H는 추론규칙으로 정의되어 있지만, 사실 임의의 B, h, h'에 대해 $B \vdash h\ \nabla_h h' \Rightarrow h''$인 h''를 항상 계산할 수 있고, 임의의 B, H에 대해 $B \vdash \nabla_H H \Rightarrow \{h\}$인 h를 항상 계산할 수 있다. widen(B, H) 그림 11은 주어진 H를 ∇_H를 이용하여 H보다 더 큰 크기 1이하인 $Histories$를 계산해 준다(참고로 크기가 0인 H는 {}인 경우이며, 주어진 H가 {}일 때만 {}로 계산된다).

위에서 설명한 trim과 widen은 팽창함수이기 때문에 기존의 알고리즘의 곳곳에 합성(composition)할 경우, 우리는 최소고정점보다 더 큰 결과를 얻게 된다. 즉, 안전한 결과를 얻는다. 알고리즘 상에서 d가 만들어지는 부분마다 그 결과에 trim을 적용시키도록 수정한다. 예를 들면, $\sqcap_d\{\cdots\}$부분이 있으면 trim($\sqcap_d\{\cdots\}$)로 수정한다. 삽입되는 각 trim이 모두 정확하게 같은 일을 할 필요는 없다. 단지 주어진 실행요약보다 큰 결과만 반환해 주면 된다. 단, 한가지 집고 넘어갈 점은 주어진 실행요약 안에 ($\top_B$, H)가 있었을 경우, 적어도 그것은 제거시키면 안 된다. 만약 이것을 제거시키는 경우에는, 분석중인 함수의 최종 결과에 $\top_B$ 가정에 대한 결과가 포함되지 않게 되며, 또 다른 함수에서 이 함수를 호출하는 경우에 이 함수의 실행요약에 나타나는 가정들을 전부다 만족시키지 못할 경우, 그 결과는 $\top_d$가 되어버려서, 쓸모 없는 분석결과를 얻게 될 수 있다. widen의 경우는 크기가 2이상인 $Histories$가 생길 때마다 widen을 통해 크기 1짜리 $Histories$로 만들게끔 수정한다(path-insensitive). 예를 들면 update(B, h, V_1, V_2)를 사용하는 대신 widen(B, update(B, h, V_1, V_2))를 사용한다.

마지막으로 수정될 부분은 바로 개선된 알고리즘의 종료를 보장하기 위한 부분이다. 현재까지의 수정만으로는 단지 원래의 알고리즘의 결과를 개선된 알고리즘의 결과가 포섭한다는 것까지만 보장이 된다. 팽창함수와 단조증가함수들의 합성함수를 $g : X \to X$라 하면, 다음

[∇_h과 ∇_H의 추론규칙]

$$\frac{}{B \vdash \epsilon \nabla_h \epsilon \Rightarrow \epsilon}$$

$$\frac{B \vdash \epsilon \nabla_h h \Rightarrow h'}{B \vdash h \nabla_h \epsilon \Rightarrow h'}$$

$$\frac{B \vdash h \sqsubseteq_h h''' \cdot [AP \mapsto V'] \quad B \vdash h''' \nabla_h h' \Rightarrow h''}{B \vdash h \nabla_h h' \cdot [AP \mapsto V] \Rightarrow h'' \cdot [AP \mapsto V \sqcup_V V']}$$

$$\frac{H = \{\} \vee H = \{h\}}{B \vdash \nabla_H H \Rightarrow H}$$

$$\frac{B \vdash \nabla_H (H \backslash H') \Rightarrow \{h\} \quad B \vdash \nabla_H H' \Rightarrow \{h'\} \quad B \vdash h \nabla_h h' \Rightarrow h''}{B \vdash \nabla_H H \Rightarrow \{h''\}}$$

[알고리즘의 개선을 위한 함수들]

$$\begin{aligned} \text{trim} &: \mathit{AbstractStatement} \rightarrow \mathit{AbstractStatement} \\ \text{trim}(d) &= \text{some } d' \text{ such that } d \sqsubseteq_d d' \text{ (본문 참조)} \end{aligned}$$

$$\begin{aligned} \text{widen} &: \mathit{Assumptions} \times \mathit{Histories} \rightarrow \mathit{Histories} \\ \text{widen}(B, H) &= \text{some } H' \text{ such that } B \vdash \nabla_H H \Rightarrow H' \end{aligned}$$

[개선된 알고리즘]

$$\begin{aligned} &I_p : \mathit{Program} \rightarrow \widehat{Env} \\ &I_p(p) = \\ &\quad \text{let } \beta = \lambda f.\overline{y_i} \text{ where } (\overline{y_i}, \_, \_) = p(f) \\ &\quad \text{let } F(\gamma) = \lambda f \in \text{dom}(p).\text{trim}(\gamma(f) \sqcup_d I_d(p(f), \beta, \gamma)) \text{ in} \\ &\quad \text{let } G(\gamma) = \text{if } F(\gamma) \sqsubseteq_\gamma \gamma \text{ then } \gamma \text{ else } G(F(\gamma)) \text{ in} \\ &\quad G(\bot_\gamma) \end{aligned}$$

그림 11 알고리즘의 개선을 위한 함수들

과 같은 나열은 체인(chain)이 아니다:

$$\bot,\ g(\bot),\ g^2(\bot),\ \cdots$$

위와 같은 나열에 대해서 $g^{i+1}(\bot) \sqsubseteq g^i(\bot)$이 되는, i가 존재한다는 것이 보장되지 않으므로, 개선된 알고리즘의 종료 또한 보장되지 않는다. 따라서 우리는 위의 나열대신 다음과 같은 나열을 생각해 보자(여기서 e는 임의의 팽창함수):

$$h_0 = \bot,$$
$$h_{i+1} = e(h_i \sqcup g(h_i))$$

$h_0 = \bot \sqsubseteq h_i$이고 $h_i \sqsubseteq h_i \sqcup g(h_i) \sqsubseteq e(h_i \sqcup g(h_i)) = h_{i+1}$ 이므로 위의 나열은 증가체인이다. 그런데 X가 유한하다면, 위의 체인 또한 유한하며 따라서 $h_{k+1} \sqsubseteq h_k$인 유한한 k가 존재한다. 그리고 사실 이러한 h_k는 g의 앞-고정점이다. 왜냐하면, $h_k \sqsupseteq h_{k+1} = e(h_k \sqcup g(h_k)) \sqsupseteq h_k \sqcup g(h_k) \sqsupseteq g(h_k)$이기 때문이다. 우리는 이 점을 이용하여, 원래의 알고리즘의 $I_p : \mathit{Program} \rightarrow \widehat{Env}$를 그림 11과 같이 수정한다. 참고로 위에서 e는 알고리즘에서 trim에 해당한다.

4. 실 험

C언어를 대상으로 CIL[27]을 이용하여, 기본적으로 3.6절에서 개선한 알고리즘을 구현하였다. C의 구조체나 공용체에 대해서 모든 필드들을 뭉쳐서(field-insensitive) 하나의 필드로 나타나도록 하였다. 실험은 Mediabench, SPEC2000, GNU-textutils의 일부 프로그램들을 대상으로 하였다.

표 1과 표 2는 실험결과를 보여준다. 표 1의 각 항목의 의미는 다음과 같다. LOC는 해당 프로그램을 CIL [9]의 merge기능을 이용하여 여러 소스파일을 하나의 파일로 합친 후, 모든 주석 및 빈 줄을 제거시킨 후 얻은 파일의 줄 수이다. Partition limit은 앞 절에서 설명한 하나의 실행요약이 가질 수 있는 가정의 최대 개수를 의미한다. 이것을 1,2,5의 제한을 두어, 한 프로그램당 3번씩 분석하였다. Time은 분석에 걸린 시간이며 Memory는 분석 도중 메모리의 최대 사용치를 나타낸다. Strong Update %는 분석 도중에 수행된 업데이트 명령 중에서 업데이트 당하는 V_1이 $\{AP\}$이고 AP가 뭉쳐진 접근경로가 아닌 경우의 비율을 나타낸다. V_1의 크기가 1이라는 것은 곧 V_1이 나타내는 추상화된 접근경로가 1개라는 것이며 그 추상화된 접근 경로가 뭉쳐지지 않았다는 것은 곧 실제로 나타내는 메모리 위치가 단 1개라는 것을 의미한다. 이런 경우 갱신 대상으로 주어진 업데이트 기록에서 기존의 V_1에 대한 업데이트 기록은 완전히 제거되고 V_2로의 업데이트 명령만 남게 되

표 1 실험 결과 1

Program	LOC	Partition limit	Time	Memory (MB)	Strong Update %	Avg dom	Avg range	Stack escape
mcf	1530	1	1s	2.2	0.8%	1.308	2.431	0
		2	1s	2.2	0.6%	1.308	2.403	0
		5	1s	2.2	0.5%	1.308	2.403	0
pr	2669	1	1s	4.7	43.9%	1.439	1.975	0
		2	1s	4.7	43.5%	1.439	1.616	0
		5	4s	4.9	47.8%	1.439	1.719	0
sort	3620	1	1s	5.2	30.9%	0.578	2.773	0
		2	2s	6.1	33.0%	0.578	2.576	0
		5	6s	7.1	33.4%	0.578	2.583	0
bzip2	3889	1	1s	6.9	94.7%	0.5	1.688	0
		2	1s	6.9	94.7%	0.5	1.688	0
		5	1s	6.9	94.7%	0.5	1.688	0
rasta	4984	1	2s	8.9	57.4%	2.451	2.762	0
		2	2s	9.3	59.9%	2.451	2.412	0
		5	5s	10.3	64.1%	2.451	2.236	0
pegwit	5707	1	2s	12.0	100.0%	0.222	1.929	0
		2	2s	12.3	100.0%	0.222	1.929	0
		5	2s	12.3	100.0%	0.222	1.929	0
ammp	13738	1	44s	30.0	48.0%	2.698	7.547	0
		2	5m42s	32.2	48.9%	2.698	7.289	0
		5	38m43s	44.7	45.9%	2.697	7.322	0
vpr	17078	1	1m9s	42.0	49.8%	2.613	5.849	0
		2	7m51s	54.8	52.3%	2.565	5.289	0
		5	52m25s	86.8	52.7%	2.563	5.211	0
crafty	22001	1	11s	40.1	82.4%	1.018	1.863	0
		2	11s	39.8	82.4%	1.018	1.863	0
		5	11s	39.8	82.4%	1.018	1.863	0
twolf	24868	1	1m20s	60.2	42.7%	2.555	4.502	0
		2	11m39s	81.1	43.7%	2.531	4.298	0
		5	103m4s	151.9	43.4%	2.536	4.239	0

표 1 실험 결과 2

Program	Partition limit								
	2			5					
	1	2	Avg	1	2	3	4	5	Avg
mcf	24	2	1.08	24	2	0	0	0	1.08
pr	32	9	1.22	32	4	4	0	1	1.39
sort	43	2	1.04	43	0	1	1	0	1.11
bzip2	74	0	1.00	74	0	0	0	0	1.00
rasta	68	3	1.04	68	1	0	0	2	1.13
pegwit	99	0	1.00	99	0	0	0	0	1.00
ammp	155	24	1.13	155	9	11	2	2	1.25
vpr	253	47	1.16	261	8	11	4	16	1.35
crafty	109	0	1.00	109	0	0	0	0	1.00
twolf	163	28	1.15	163	5	6	9	8	1.40

기 때문에, 분석의 정확도가 높아진다는 점에서 의미가 있는 수치이다. 한편 프로그램의 각 함수마다 하나씩의 실행요약(*AbstractStatement*)이 생성되는데, 각 실행요약은 가정과 업데이트 기록 집합의 짝들로 이루어져 있다. 먼저 각 실행요약마다 평균적인 업데이트 명령 개수를 구한 뒤, 모든 실행요약에 대해서 다시 평균을 낸 결과가 Avg dom이다. 또한 먼저 각 실행요약 내의 각 업데이트 $[AP \rightarrow V]$에 들어있는 V가 나타내는 접근 경로의 개수(즉, $|V|$)의 평균을 낸 뒤, 마찬가지로 모든 실행요약에 대해서 다시 평균을 낸 결과가 Avg range이다. 따라서 Avg dom과 Avg range가 낮을수록 정확한 결과라고 할 수 있다. 마지막으로 Stack escape는, 매개변수나 지역변수의 주소가 함수 밖으로 유출될 수 있다는 경고의 개수이다. 각 함수마다 실행요약을 계산해 낸 뒤에 알고리즘대로 지역변수나 매개변수를 실행요약에서 제거하는데, 이때 만약 $[AP \rightarrow V]$와 같은 업데이트 명령에서 AP가 지역변수나 매개변수가 아니고 V에 어떤 지역변수나 매개변수가 들어있다면, 이것은 곧 해당 지역변수나 매개변수의 주소가 함수바깥으로 유출될 수 있다는 것을 의미한다. 이런 경우의 개수를 센 것이

Stack escape이다.

표 2는 두 개 이상의 가정을 갖는 실행요약이 많을수록, 가정을 많이 필요로 하는(가정을 잘 활용하는) 프로그램일 것이라는 추측에서 측정해 본 항목이다. 표 2는 하나의 실행요약이 가질 수 있는 가정의 최대 개수를 2와 5로 설정한 상태에서, 프로그램 상의 각 함수의 실행요약이 가지는 가정의 개수를 나타낸 것이다. 예를 들면, pr의 경우 한 실행요약이 가지는 가정의 최대개수를 2로 제한한 상태에서 분석했을 때, 총 41개의 함수 중에서 32개의 함수의 실행요약은 단 한 개의 가정만을 가졌고, 나머지 9개의 함수의 실행요약은 두 개의 가정을 가졌다. Avg는 각 함수의 실행요약에 있는 가정의 개수의 평균을 나타낸다.

실험결과를 살펴보면, 가정의 최대 개수제한에 상관없이 같은 결과를 내는 프로그램들도 있고, 개수제한이 늘어남에 따라 다른 결과를 내는 프로그램들도 있다. 표 1과 표 2를 함께 살펴보면, 가정의 활용도가 상대적으로 높은 프로그램들이 후자에 속한다는 것을 알 수 있다. 한편, ammp, vpr, twolf와 같은 프로그램들은 가정의 개수 제한이 늘어남에 따라 분석에 걸리는 시간이 급격하게 증가하는데, 이것은 가정의 활용도가 높아서(표 2 참조) 실질적으로 분석에 사용되는 의미공간이 그렇지 않은 프로그램들보다 커진 효과가 발생하여 고정점에 도달하는데 그만큼 시간이 오래 걸리게 되는 것으로 추측된다. 이렇게 시간이 너무 오래 걸리는 경우를 위해, 정확도를 약간 희생함으로써 고정점에 더욱 빨리 도달하게끔 하는 휴리스틱이 필요한 것으로 보인다.

5. 결론 및 향후 연구

포인터가 있는 언어로 작성된 프로그램을 분석하기 위해서는, 일반적으로 포인터 분석의 선행이 필요하다. 우리는 업데이트 기록에 기반하여 상향방식의 프로그램 흐름과 호출 문맥에 민감한 포인터 분석 알고리즘을 설계하였다. 업데이트 기록은 상향방식 분석을 정형화 할 수 있게 해 주었을 뿐만 아니라, 죽은 메모리 반응과, 관련 별칭 문맥을 찾는 데에도 활용 되었다. 그리고 이 알고리즘을 포인터가 있는 대표적인 언어인 C언어에 대해 구현하였다. C언어의 여러 기능들을 지원하기 위해, 그리고 분석에 걸리는 시간을 줄이기 위해 알고리즘을 확장하였다. 또한, 여러 C언어 프로그램에 대해, 포인터 분석기가 분석하는데 걸리는 시간, 필요한 메모리, 생성된 요약함수들의 특성 등을 측정해 보았다.

포인터 분석이 상향단계만을 가지도록 설계함으로써 여러 장점을 가지게 되었다. 즉, 어떤 모듈을 분석 시에 하위 모듈 함수들의 소스코드 대신 실행요약을 사용하고 상위 모듈들에 대한 정보가 전혀 없이도 분석할 수 있게끔 설계되었다. 이런 구조 덕분에 프로그램의 시작점이 존재하지 않는 상향방식으로 작성되고 있는 프로그램의 개발 중에 분석을 수행할 수 있다는 장점과 프로그램의 한 모듈이 수정될 경우, 그 하위 모듈들은 다시 분석될 필요가 없다는 장점을 가지게 되었다.

차후 연구 방향은 제안된 분석기의 안전성 및 끝남을 증명하고 본 논문에서 제안된 방식과 1.3절에서 이론적으로 비교된 기존의 상향 방식의 분석을 실험을 통해 비교해 보는 일이다. 또한, 제한된 수 내에서 가정들을 효율적으로 제어하는 방법에 대해서도 알아보고자 한다. 이 분석기의 실용성을 보이기 위하여 구현된 포인터 분석기를 Pruv-C[28]와 같은 상향방식의 프로그램 분석과 연동하여 실제로 사용할만한 성능과 정확도를 내는지 살펴볼 것이다.

참 고 문 헌

[1] D. R. Chase, M. Wegman, F. K. Zadeck, Analysis of pointers and structures, in: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1990, pp. 296-310.

[2] W. Landi, B. G. Ryder, A safe approximate algorithm for interprocedural pointer aliasing, in: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1992, pp. 235-248.

[3] A. Deutsch, Interprocedural may-alias analysis for pointers: Beyond k-limiting, in: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1994, pp. 230-241.

[4] M. Hind, M. Burke, P. Carini, J. D. Choi, Interprocedural pointer alias analysis, ACM Transactions on Programming Language and Systems (1999) 848-894.

[5] M. Emami, R. Ghiya, L. J. Hendren, Context-sensitive interprocedural points-to analysis in the presence of function pointers, in: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1994, pp. 242-256.

[6] R. P. Wilson, M. S. Lam, Efficient context-sensitive pointer analysis for C programs, in: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1995, pp. 1-12.

[7] R. P. Wilson, Efficient, context-sensitive pointer analysis for c programs, Ph.D. thesis, Stanford University (Dec. 1997).

[8] M. Sagiv, T. Reps, R. Wilhelm, Parametric shape analysis via 3-valued logic, ACM Transactions on Programming Language and Systems (2002) 217-

298.
[9] J. Zhu, Towards scalable flow and context sensitive pointer analysis, in: *Proceedings of ACM Design Automation Conference*, 2005, pp. 831-836.
[10] N. Rinetzky, M. Sagiv, E. Yahav, Interprocedural shape analysis for cutpoint-free programs, in: C. Hankin, I. Siveroni (Eds.), SAS, Vol. 3672 of Lecture Notes in Computer Science, Springer, 2005, pp. 284-302.
[11] N. Rinetzky, J. Bauer, T. W. Reps, S. Sagiv, R. Wilhelm, A semantics for procedure local heaps and its abstractions, in: J. Palsberg, M. Abadi (Eds.), POPL, ACM, 2005, pp. 296-309.
[12] A. Gotsman, J. Berdine, B. Cook, Interprocedural shape analysis with separated heap abstractions, in: K. Yi (Ed.), SAS, Vol. 4134 of Lecture Notes in Computer Science, Springer, 2006, pp. 240-260.
[13] D. Distefano, P. W. O'Hearn, H. Yang, A local shape analysis based on separation logic, in: H. Hermanns, J. Palsberg (Eds.), TACAS, Vol. 3920 of Lecture Notes in Computer Science, Springer, 2006, pp. 287-302.
[14] M. Fähndrich, J.Rehof, M. Das, Scalable context-sensitive flow analysis using instantiation constraints, in: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000, pp. 253-263.
[15] J. S. Foster, M. Fähndrich, A. Aiken, Polymorphic versus monomorphic flow-insensitive points-to analysis for C, in: *Proceedings of the Annual International Static Analysis Symposium*, 2000, pp. 175-198.
[16] B.-C. Cheng, W. mei W. Hwu, Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation, in: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000, pp. 57-69.
[17] E. Ruf, Effective synchronization removal for java, in: *Proceedings of the ACM SIGPLAN Conferenceon Programming Language Design and Implementation*, 2000, pp. 208-218.
[18] N. Heintze, O. Tardieu, Ultra-fast aliasing analysis using CLA: A million lines of C code in a second, in: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2001, pp. 254-263.
[19] A. Rountev, B. G. Ryder, Points-to and side-effect analyses for programs built with precompiled libraries, in: *Proceedings of International Conference on Compiler Construction*, 2002, pp. 20-36.
[20] M. J. Harrold, G. Rothermel, Separate computation of alias information for reuse, IEEE Transactions on Software Engineering (1996) 442-460.
[21] A. Rountev, B. G. Ryder, W. Landi, Data-flow analysis of program fragments, in: *ACM SIGPLAN-SIGSOFT Symposium on the Foundations of Software Engineering*, 1999, pp. 235-252.
[22] J. Whaley, M. C. Rinard, Compositional pointer and escape analysis for java programs, in: OOPSLA, 1999, pp. 187-206.
[23] R. Chatterjee, B. G. Ryder, W. A. Landi, Relevant context inference, in: *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1999, pp. 133-146.
[24] M. Buss, D. Brand, V. C. Sreedhar, S. A. Edwards, Flexible pointer analysis using assign-fetch graphs, in *Proceedings of the ACM Symposium on Applied Computing*, 2008, pp. 234-239.
[25] C. Calcagno, D. Distefano, P. W. O'Hearn, H. Yang, Footprint Analysis, in: *Proceedings of Static Analysis Symposium*, 2007, 402-418.
[26] P. Cousot, R. Cousot, Abstract interpretation frameworks, Journal of Logic and Computation 2 (4) (1992) 511-547.
[27] G. C. Necula, S. McPeak, S. P. Rahul, W. Weimer, CIL: Intermediate languages and tools for analysis and transformation of C programs, in: *Proceedings of International Conference on Compiler Construction*, 2002, pp. 213-228.
[28] H.-G. Kang, Y. Kim, T. Han, H. Han, A path sensitive type system for resource usage verification of C like languages, in: K. Yi (Ed.), APLAS, Vol. 3780 of Lecture Notes in Computer Science, Springer, 2005, pp. 264-280.

박 상 운
2001년~2005년 한국과학기술원 전자전산학과 전산학 전공(학사). 2005년~2007년 한국과학기술원 전자전산학과 전산학 전공(석사). 2007년~현재 LG전자 디지털미디어 연구소. 관심분야는 프로그램 분석, 임베디드 시스템 설계 및 분석

강 현 구
1991년~1997년 한양대학교 전산학과(학사). 1997년~1999년 한양대학교 전산학과(석사). 1999년~2000년 한국전자통신연구원 연구원. 2001년~현재 한국과학기술원 전산학과 박사과정. 관심분야는 프로그램 분석, 소프트웨어 공학, 임베디드 시스템 설계 및 분석

한 태 숙
정보과학회논문지 : 소프트웨어 및 응용
제 35 권 제 7 호 참조