
ROPAS

Research On Program Analysis System
National Creative Research Initiative Center 1998-2003
Programming Research Laboratory, School of Computer Science & Engineering
Seoul National University

ROPAS MEMO
ROPAS-2005-27
October 27, 2005

C 프로그램의 배열 참조 오류(buffer overrun)를 모두 자동으로 찾아 주는 정적 분석기의 설계와 구현

정영범, 김재황, 신재호, 이광근
{dreameye,jaehwang,netj,kwang}@ropas.snu.ac.kr
서울대학교 컴퓨터 공학부 / 프로그래밍 연구실

October 27, 2005

Abstract

이 논문에서는 정적 프로그램 분석(static program analysis)과 통계적 분석(statistical analysis)의 조합을 가지고 실제적인 C 프로그램을 검증한 경험을 보고한다. 우리는 ANSI C 프로그램이 실행 중에 겪을 수 있는 모든 배열 참조 오류(buffer overrun)를 찾아주는 정적 프로그램 분석기 아이락(Airac, Array Index Range Analyzer for C)을 고안하고 구현하였다. 분석의 안전성(soundness)을 유지하면서 프로그램 분석 분야에서 오랫동안 축적된 기술들을 활용하여 분석 비용 절감 및 정확도 향상을 달성했다. 대상 프로그램에 제한을 두지 않는 안전한 분석은 정확도에 한계가 있으므로 불가피한 허위 경보(false alarm)가 존재할 수 있다. 이러한 허위 경보에 대처하기 위해 우리는 통계적 사후 분석을 시도하였다. 통계적 사후 분석을 통해 주어진 경보가 실제 오류를 가리킬 확률을 계산한다. 이렇게 계산된 확률을 이용해서 경보를 걸러내거나 실제 오류에 대한 경보일 가능성이 높은 것들을 우선적으로 사용자에게 보여주도록 하였다. 우리는 리눅스 커널 프로그램과 알고리즘 교과서의 프로그램들을 대상으로 실험을 수행했다. 우리 실험에서 실제 오류를 놓칠 위험이 허위 경보의 위험의 3배라고 설정한 경우 74.83%의 허위 경보를 걸러낼 수 있었고, 참일 확률이 높은 경보부터 검증하는 경우 15.17%의 허위 경보만이 실제 오류 50%와 섞여 있었다.

1 서론

할당된 배열의 범위를 벗어나는 주소를 참조할 때 발생하는 배열 참조 오류(buffer overrun)는 C 프로그램의 오래고도 흔한 문제이다. 배열 참조 오류(buffer overrun)는 프로그램의 비정상적인 종료뿐 아니라 보안 상의 허점[1]으로 작용하여 심각한 문제를 야기하기도 한다. 일반적으로 프로그램의 입력값의 범위는 무한하므로 테스트를 통해서는 배열 참조 오류를 완벽하게 제거할 수가 없다. 한편 Java 언어와 같이 실행 중에 배열 참조가 올바른 지를 검사하는 방법은 프로그램을 안전하게 종료시킬 수는 있으나 배열 참조 오류를 제거할 수는 없다. 게다가 실행 중 검사는 프로그램 실행의 비용을 증가시키는 문제를 야기한다.

배열 참조 오류(buffer overrun)에 대한 최선의 해결책은 프로그램 실행 전에 발생 가능한 모든 배열 참조 오류(buffer overrun)를 찾아서 제거하는 것이다. 이것이 가능하다면 실행 중

[1] 1/2[2]에서 2/3[1]의 보안 문제가 배열 참조 오류와 관련이 있다.

배열 참조 검사로 인한 부담을 덜 수 있으며, 테스트로부터는 얻을 수 없는 "오류 없음"에 대한 확신을 가질 수도 있다.

아이락(Airac, Array Index Range Analyzer for C)은 정적 프로그램 분석(static program analysis) 기술을 이용하여 C 프로그램에 잠재된 모든 배열 참조 오류(buffer overrun)를 프로그램의 실행 전에 자동으로 찾아주는 분석기이다. 아이락의 설계는 세가지 목표를 달성하는 데 초점을 맞춰 이루어졌다: 아이락은 1) 입력 프로그램이 실행 중에 겪을 수 있는 모든 상황을 포섭하는 분석을 수행하여 발생할 수 있는 모든 배열 참조 오류를 찾아 줄 수 있어야 하며; 2) 분석 비용과 분석 정확도 간의 균형 잡힌 분석을 사용자에게 제공해야 하며; 3) 프로그램의 응용 영역(application domain)이나 개발 환경 등에 대한 가정을 하지 않아야 한다. 다시말해 특정 스타일을 가진 프로그램들에 최적화된 분석이 아니라 임의의 C 프로그램을 대상으로 하는 분석을 제공해야 한다. 실제적인 C 프로그램에 대해 앞의 세 가지 조건을 만족하는 배열 참조 오류(buffer overrun) 분석기를 만드는 일은 결코 간단하지 않다. 기존의 연구들[9, 4, 15, 11]을 살펴 보면 세 가지 조건 중 하나 이상은 포기하고 있다. 분석 대상 프로그램의 특성에 대한 제한을 두지 않으면서 안전한 분석(sound analysis)을 하면 불가피하게 허위 경보(false alarm)들이 많이 발생하게 된다. 허위 경보(false alarm)란 실제 오류가 아님에도 분석의 정확도가 떨어져서 오류라고 경보를 하는 것을 말한다. 안전한 분석에 있어 가장 중요한 문제는 불가피한 허위 경보(false alarm)들을 얼마나 효과적으로 줄일 수 있는가 하는 것이다.

허위 경보 문제에 대응하기 위해 우리는 다음과 같은 전략을 취했다: 적절한 수준의 비용을 지불하는 선까지 분석 정확도를 향상시킬 수 있도록 분석 설계를 하고, 통계 기법을 이용한 사후 처리를 통해 허위 경보 문제를 해결한다. 아이락은 ANSI C 프로그램에서 실행 중에 배열 참조 오류(buffer overrun)가 발생하는 곳을 모두 자동으로 찾아 준다. 아이락은 분석의 안전성(soundness)을 지키면서 프로그램 분석(program analysis) 분야에 오랫동안 축적된 기술들을 활용해 분석 비용과 정확도 사이의 균형을 맞추었다. 하지만 정적 프로그램 분석에 대해서는 허위 경보를 발생시키는 프로그램이 항상 존재한다. 이러한 허위 경보들을 다루기 위해 통계적 사후 분석을 이용하였다. 통계적 사후 처리는 분석 결과를 받아 각 경보가 참일 확률을 계산한다. 계산된 확률은 두 가지 방법으로 허위 경보 문제에 적용될 수 있다: 1) 경보를 거르는 데 사용할 수 있다. 사용자가 지정한 임계치를 기준으로 임계치 이상이 되는 확률을 가지는 경보들만 사용자에게 보고한다; 2) 경보의 보고 순서 조정에 적용할 수 있다. 실제 오류를 가리킬 확률이 높은 경보들을 먼저 보고하여 사용자로 하여금 경보 검증을 보다 효율적으로 할 수 있도록 한다.

아이락은 ANSI C 언어에서 사용가능한 모든 배열 참조 방식을 분석해 낼 수 있다. 간단한 산수식에서 함수 호출을 포함하는 임의의 프로그램 식을 비롯하여 포인터 연산, 별명(alias)에 의한 참조 등 모든 경우를 포괄하는 분석을 수행한다. 배열의 경우도 정적으로 할당된 경우나 동적으로 크기가 결정되어 할당된 경우 모두 분석할 수있다.

분석의 비용과 정확도 간의 균형을 맞추기 위해서 다음의 기술들이 사용되었다. 분석 정확도 향상을 위해서는 축지법(widening) 후에 좁히기(narrowing) 적용, 실행 흐름 별 분석(flow sensitivity), 함수 호출 별 분석(polyvariance), 가지치기(context pruning)[2], 정적 반복문 펼치기(static loop unrolling) 등의 기술들이 이용되었다. 분석 비용 절감을 위해서는 데이터 스택

[2]실행 과정 분할하기(trace partitioning)[13]의 한 예.

$$
\begin{array}{rcl@{\qquad}rcl}
pgm & \rightarrow & decl^* \; e & e & \rightarrow & n \mid e\texttt{+}e \\
decl & \rightarrow & vdecl \mid fdecl & & \mid & \texttt{[}e\texttt{]} \mid e\texttt{[}e\texttt{]} \\
vdecl & \rightarrow & \texttt{var}\ x \mid \texttt{var}\ x\texttt{[}e\texttt{]} & & \mid & x\texttt{:=}e \mid \texttt{*}e\texttt{:=}e \mid e\texttt{[}e\texttt{]:=}e \\
fdecl & \rightarrow & f\texttt{(}x\texttt{)\{}vdecl^*\ e\texttt{\}} & & \mid & e\texttt{;}e \mid \texttt{if}\ e\ e\ e \mid \texttt{while}\ e\ e \mid e\ e \mid \texttt{return}\ e
\end{array}
$$

Figure 1: 분석용 프로그래밍 언어 C'

무시하기(stack obviation), 선택적 합치기(selective join[3]), 분석 지연 시키기(wait-at-join) 등의 기술을 사용하였다. 530만 줄 크기의 상용 C 프로그램에 대한 실험 결과 아이락은 970개의 경보를 발생시켰으며 그 중 233개의 경보가 실제 배열 참조 오류(buffer overrun)에 대한 것이었다. 일부 리눅스 커널(Linux kernel) 프로그램에 대한 실험에서는 총 18,760 줄의 코드에서 26개의 경보를 발생시켰으며 그중 16개가 실제 배열 참조 오류(buffer overrun)를 가리키는 것이었다.

허위 경보 문제는 몬테카를로 기법(Monte Carlo method)[14]과 결정 이론(decision theory)[3]을 적용하여 구현된 베이지안 통계 분석(Bayesian statistical analysis)[10]으로 접근해 보았다. 우리는 주어진 경보들에서 관측할 수 있는 증상(symptom)들로 부터 경보가 참일 확률을 계산해내는 조건 확률식을 정의하였다. 이 확률의 계산에는 참/거짓으로 미리 분류된 경보들로 만들어진 교육 집합(learning set)에서 얻은 계수들이 이용 된다. 계수들은 몬테카를로 기법을 이용해 계산된다. 실험에 사용된 교육집합은 리눅스 커널, 알고리즘 교과서 등에서 얻은 C 프로그램들을 분석하여 경보의 참/거짓 여부를 검증한 후 만든 것이다. 각 경보가 참일 확률을 계산한 후에는 사용자가 지정한 임계치를 기준으로 임계치 이상의 확률을 가지는 경보들만 사용자에게 보고할 수 있다. 임계치는 허위 경보를 보고 했을 때의 위험치에 대한 실제 오류를 보고 하지 않았을 때의 위험치의 비로 결정된다. 실제 오류를 보고하지 않을 위험이 허위 경보를 발생시키는 위험의 3배라고 했을 때, 우리의 실험에서 73.83%의 허위 경보를 제거할 수 있었다. 확률을 경보 보고 순서 결정에 적용했을 때는 사용자가 실제 오류의 50%를 확인 하는 동안 15.17%의 허위 경보만을 사용자가 보게되는 결과를 얻었다.

2 분석기 설계

아이락은 요약해석 틀(abstract interpretation framework)[5, 6]을 따라 설계되었다. 모든 발생 가능한 배열 참조 오류를 찾기 위해 아이락은 프로그램의 각 지점에서 프로그램의 실행 중에 일어날 수 있는 상태들의 안전한 요약(sound approximation)을 계산 한다. 안전한 요약이란 실제 일어날 모든 일을 모두 포섭하는 것을 말한다. 요약 해석의 틀에서 분석이 안전함을 보이기 위해서는 요약 함수(abstraction) α와 구체화 함수(concretization) γ로 구성된 갈로아 연결(Galois connection)이 존재하는 실제 의미 공간(concrete semantic domain)과 요약 의미 공간(abstract semantic domain)이 있어야한다. 그리고 실제 의미 공간에서 정의된 실제 의미

[3]join연산은 최소상계 연산(least upper bound)연산 이다.

함수 $\mathcal{F}$와 요약 의미 공간에서 정의된 요약 의미 함수 $\widehat{\mathcal{F}}$는 다음의 관계 를 만족해야 한다:

$$\begin{aligned} \alpha \circ \mathcal{F} &\sqsubseteq \widehat{\mathcal{F}} \circ \alpha \\ \mathcal{F} \circ \gamma &\sqsubseteq \gamma \circ \widehat{\mathcal{F}} \end{aligned}$$

이번 장에서는 아이락의 설계에서 사용된 실제 의미 공간, 요약 의미 공간 그리고 각 공간에서의 의미 함수들을 제시한다. 그리고 요약 의미 함수의 고정점을 구하는 알고리즘을 설명한다.

의미 공간 및 실행 의미를 살펴보기 전에 아이락의 분석용 언어인 C'을 먼저 알아보자. 아이락이 사용하는 의미 공간 및 실행 의미 정의는 C' 언어를 대상으로 이루어 졌다. 아이락은 C로 작성된 프로그램을 분석하기 전에 분석용 언어인 C'으로 변환 한다. 그림 1은 배열에 관련된 부분만 추린 C' 언어의 요약 구문(abstract syntax)을 보여준다. C' 언어는 C 언어에서 설탕 구조(syntactic sugar)를 제거하고 남은 필수적인 프로그램 구성자(program construct)만을 가지며, 명확한 실행 의미(semantics)가 정의되어있다.

2.1 실제 실행 의미

C'의 실제 실행 의미는 기계 상태(*Machine*)의 전이(transition) $\rightsquigarrow$로 정의 된다:

$$\langle c, x \rangle \rightsquigarrow \langle c', x' \rangle$$

여기서 c, c'는 C의값이며, x, x'는 $State$의 값이다. $\rightsquigarrow$는 2.2 절에 정의되어있다. C'의 실제 실행 의미를 정의하기 위한 실제 의미 공간(concrete semantic domain)은 그림 2에 정의되어 있다. 기계 상태는 명령문(C)과 상태($State$)로 구성된다. 상태 $State$는 데이터 스택($Stack$), 환경(Env), 메모리(Mem), 제어 스택($Dump$)의 데카르트 곱(Cartesian product)으로 정의된다. 데이터 스택 $Stack$은 프로그램 식(expression)의 값을 저장하기 위해 프로그램 식의 레이블(label)과 그 프로그램 식의 값으로 구성된 짝(pair)들의 리스트 $(Lab \times Val)^*$로 정의된다. 환경은 식별자(identifier)를 메모리 주소에 대응시키는 맵 $Id \xrightarrow{\text{fin}} Addr$으로 정의된다. 메모리는 주소에서 값을 대응시키는 맵 $Addr \xrightarrow{\text{fin}} Val$로 정의된다. 명령문 C는 원래 프로그램에 있던 프로그램 식, 선언, 전이 과정에서 생성되는 지시문(instruction)들로 구성된다. 제어 스택 $Dump$는 함수 호출과 함수 반환(return)을 위해 필요하다. 제어 스택 $Dump$에는 함수가 호출될 때의 환경, 함수 반환 후에 실행해야할 명령문, 함수의 반환 값에 해당하는 데이터 스택의 레이블들이 저장된다. 이를 위해 $Dump$는 $(Lab \times C \times Env)^*$로 정의한다. C' 프로그램이 실행 중에 만드는 값의 공간은 정수 값과 배열을 가리키는 주소(Blk), 함수 값으로 구성된다. Blk은 배열의 시작 주소, 오프셋(offset), 크기로 이루어졌다.

프로그램의 분석을 위한 실행 의미(collecting semantics)는 프로그램의 실제 실행 과정(trace)을 모두 모으는 것이다. 이는 프로그램 포인트와 그 곳에서의 상태 $Edge \times State$의 전이열(transition sequence)로 정의된다. 그러므로 프로그램의 실제 실행 의미는 다음 함수의 최소 고정점(least fixpoint)으로 정의된다:

$$\begin{aligned} \mathcal{F} &: 2^{(Edge \times State)^*} \to 2^{(Edge \times State)^*} \\ \mathcal{F}(X) &= \{m_0\} \cup \{t \cdot \langle \langle l', l'' \rangle, s' \rangle \mid t \overset{\text{let}}{=} \cdots \langle \langle l, l' \rangle, s \rangle \in X, \langle c_{l'}, s \rangle \rightsquigarrow \langle c_{l''}, s' \rangle\} \end{aligned}$$

$$
\begin{array}{rrcl}
x, f \in & Id & & \\
l, L \in & Lab & & \\
v \in & Val & = & \mathbb{Z} + Closure + Blk + \{\bot\} \\
e \in & Exp & & \\
d \in & Dec & & \\
m, n \in & \mathbb{Z} & & \\
a \in & Addr & = & \mathbb{Z} \\
 & Blk & = & Addr \times \mathbb{Z} \times \mathbb{Z} \\
M \in & Mem & = & Addr \xrightarrow{\text{fin}} Val \\
\sigma \in & Env & = & Id \xrightarrow{\text{fin}} Addr \\
 & Fexpr & = & \{f\lambda x.e, \dots\} \\
 & Closure & = & Fexpr \times Env \\
\texttt{bop} \in & Binop & = & \{\texttt{add}, \texttt{minus}, \texttt{mul}, \texttt{divide}, \texttt{lt}, \texttt{equal}\} \\
 & Instr & = & Binop \cup \\
 & & & \{\texttt{app}, \texttt{cond}, \texttt{loop}, \texttt{ret} \\
 & & & \texttt{allocar}, \texttt{aval}, \texttt{fetch}, \\
 & & & \texttt{assign}, \texttt{vassign}, \texttt{aassign}, \} \\
c \in & C & = & \{Instr + Pgm + Dec + Exp\}^* \\
D \in & Dump & = & (Lab \times C \times Env)^* \\
S \in & Stack & = & (Lab \times Val)^* \\
\langle M, C, S, \sigma, D\rangle \in & Machine & = & C \times State \\
 & State & = & Stack \times Env \times Mem \times Dump
\end{array}
$$

Figure 2: 실제 의미 공간

레이블(label)은 프로그램의 식(expression) 및 선언(declaration)에 고유한 값으로 할당되고, 간선 $Edge = Lab \times Lab$은 연이어 실행되는 프로그램 식 및 선언의 레이블들의 쌍으로 만들어진다.

2.2 기계 상태 전의 규칙

$State = Stack \times Env \times Mem \times Dump$이므로 기계 상태 전이 $\rightsquigarrow$는 다음과 같이 정의할 수 있다:

$$\langle c, S, \sigma, M, D\rangle \rightsquigarrow \langle c', S', \sigma', M', D'\rangle$$

여기서 $c, c' \in C$, $S, S' \in State$, $\sigma, \sigma' \in Env$, $M, M' \in Mem$, $D, D' \in Dump$이다. 이 규칙이 의미하는 바는 주어진 상태 $\langle S, \sigma, M, D\rangle$를 가지고 프로그램을 명령문 c를 실행하면 다음에 할 명령문 c'와 상태$\langle S', \sigma', M', D'\rangle$가 결정된다는 것이다.

2.2.1 선언

$$
\begin{array}{lrcl}
\text{변수} & \langle x_l.c, S, \sigma, M, D\rangle & \leadsto & \langle c, S, \sigma[a/x], M[\bot/a], D\rangle \\
 & & & \texttt{allocated}(l) = \texttt{allocated}(l) \cup \{a\} \\
\text{배열} & \langle x\texttt{[}n\texttt{]}_l.c, S, \sigma, M, D\rangle & \leadsto & \langle c, S, \sigma[a/x], M[\langle a', 0, n\rangle/a, \bot/a', \ldots, \bot/a'+n-1], D\rangle \\
 & & & \texttt{allocated}(l) = \texttt{allocated}(l) \cup \{a, a', \ldots, a'+n-1\} \\
\text{함수} & \langle f\texttt{(}x\texttt{)=}e.c, S, \sigma, M, D\rangle & \leadsto & \langle c, S, \sigma[a/f], M[\langle f\texttt{(}x\texttt{)=}e, \sigma\rangle/a], D\rangle \\
 & & & \texttt{allocated}(l) = \texttt{allocated}(l) \cup \{a\}
\end{array}
$$

변수, 배열, 함수 선언은 각각의 식별자에 주소 a를 할당하고 환경 σ에 반영한다. a는 지금까지 사용된 적이 없는 새로운 주소 값이다. 변수 및 배열의 경우 메모리는 $\bot$으로 초기화되고 함수의 경우 함수 정의가 메모리에 저장된다. 선언에 달린 레이블이 l인 경우 l에서 할당된 주소들의 집합인 $\texttt{allocated}(l)$에 a를 추가한다. 배열인 경우 각 원소에 대한 주소도 $\texttt{allocated}(l)$에 추가된다. allocated는 요약 의미 실행 의미 정의에서 이용된다.

2.2.2 프로그램 식

메모리 할당

$$
\begin{array}{lrcl}
\text{배열} & \langle \texttt{[}e\texttt{]}_l.C, S, \sigma, M, D\rangle & \leadsto & \langle e.\texttt{allocar}(l).C, S, \sigma, M, D\rangle \\
\text{배열 할당} & \langle \texttt{allocar}(l).C, \langle l', n\rangle.S, \sigma, M, D\rangle & \leadsto & \langle C, \langle l, \langle a, 0, n\rangle\rangle.S, \sigma, M[\bot/a, \ldots, \bot/a+n-1], D\rangle \\
 & & & \texttt{allocated}(l) = \texttt{allocated}(l) \cup \{a, \ldots, a+n-1\}
\end{array}
$$

동적으로 배열이 할당되는 경우에는 배열 값(Blk)이 배열 할당식 $\texttt{[}e\texttt{]}$의 값이 되어 데이터 스택 S에 추가된다. 배열 원소 값들은 $\bot$으로 초기화 된다. 메모리 할당에 사용된 주소 $a, a+1, \ldots, a+n-1$는 모두 이전에 사용된 적이 없는 값이어야 한다.

값

$$
\begin{array}{lrcl}
\text{상수} & \langle v_l.C, S, \sigma, M, D\rangle & \leadsto & \langle C, \langle l, v\rangle.S, \sigma, M, D\rangle \\
\text{이항 연산} & \langle e_{l_1}\ \mathrm{bop}_l\ e_{l_2}.C, S, \sigma, M, D\rangle & \leadsto & \langle e_{l_1}.e_{l_2}.\texttt{bop}(l).C, S, \sigma, M, D\rangle \\
 & \langle \texttt{bop}(l).C, \langle l_1, v_1\rangle.\langle l_2, v_2\rangle.S, \sigma, M, D\rangle & \leadsto & \langle C, \langle l, v_1\ \mathrm{bop}\ v_2\rangle.S, \sigma, M, D\rangle \\
\text{포인터} & \langle (\texttt{*}e)_l.C, S, \sigma, M, D\rangle & \leadsto & \langle e.\texttt{deref}(l).C, S, \sigma, M, D\rangle \\
\text{포인터 값} & \langle \texttt{deref}(l).C, \langle l', \langle a, n_{\mathrm{off}}, n_{\mathrm{size}}\rangle\rangle.S, \sigma, M, D\rangle & \leadsto & \langle C, \langle l, M(a+n_{\mathrm{off}})\rangle.S, \sigma, M, D\rangle \\
 & \langle \texttt{deref}(l).C, \langle l', \langle f(x)=e, \sigma'\rangle\rangle.S, \sigma, M, D\rangle & \leadsto & \langle C, \langle l, \langle f(x)=e, \sigma'\rangle\rangle.S, \sigma, M, D\rangle \\
\text{변수} & \langle x_l.C, S, \sigma, M, D\rangle & \leadsto & \langle C, \langle l, M(\sigma(x))\rangle.S, \sigma, M, D\rangle \\
\text{배열} & \langle e_{l_1}\texttt{[}e_{l_2}\texttt{]}_l.C, S, \sigma, M, D\rangle & \leadsto & \langle e_{l_1}.e_{l_2}.\texttt{aval}(l).C, S, \sigma, M, D\rangle \\
 & \langle \texttt{aval}(l).C, \langle l_2, n\rangle.\langle l_1, \langle a, n_{\mathrm{off}}, n_{\mathrm{size}}\rangle\rangle.S, \sigma, M, D\rangle & \leadsto & \langle C, \langle l, M(a+n+n_{\mathrm{off}})\rangle.S, \sigma, M, D\rangle
\end{array}
$$

값만 계산하는 프로그램 식의 값은 계산된 후 데이터 스택에 쌓인다. 이항 연산의 경우 데이터 스택의 맨 위에 있는 값과 그 다음의 값을 이항 연산의 피 연산자로 하여 값을 계산한다. 이항 연산은 통상의 사칙 연산 및 C 언어의 연산 정의를 따른다. 다만 Blk 값에 대한 정수를 더하고 빼는 연산, 즉 포인터 연산(pointer arithmetics)은 다음과 같이 계산된다:

$$\langle a, n_{\mathrm{off}}, n_{\mathrm{size}}\rangle \pm n \quad = \quad \langle a, n_{\mathrm{off}} \pm n, n_{\mathrm{size}}\rangle$$

포인터가 가리키는 값($*e$)또는 배열 원소(e[e])의 경우 포인터 연산을 수행해서 메모리 주소를 찾은 후, 메모리에서 그 주소에 저장된 값을 꺼내 데이터 스택에 쌓는다. 실제 실행 의미에서 주소는 정수 값을 가지므로 메모리 주소 계산은 배열 시작 주소에 오프셋(offset)을 더하는 것으로 이루어진다.

대입

변수	$\langle x\texttt{=}_l e.C, S, \sigma, M, D\rangle$	$\leadsto$	$\langle e.\texttt{vassign}(l, x).C, S, \sigma, M, D\rangle$
변수 대입	$\langle \texttt{vassign}(l, x).C, \langle l', v\rangle.S, \sigma, M, D\rangle$	$\leadsto$	$\langle C, \langle l, v\rangle.S, \sigma, M[v/\sigma(x)], D\rangle$
포인터	$\langle (*e)_{l_1}\texttt{=}_l e_{l_2}.C, S, \sigma, M, D\rangle$	$\leadsto$	$\langle e_{l_2}.e_{l_1}.\texttt{assign}(l).C, S, \sigma, M, D\rangle$
포인터 대입	$\langle \texttt{assign}(l).C, \langle l_1, \langle a, n, m\rangle\rangle.\langle l_2, v\rangle.S, \sigma, M, D\rangle$	$\leadsto$	$\langle C, \langle l, v\rangle.S, \sigma, M[v/a+n], D\rangle$
배열	$\langle e_1\texttt{[}e_2\texttt{]=}_l e_3.C, S, \sigma, M, D\rangle$	$\leadsto$	$\langle e_3.e_1.e_2.\texttt{aassign}(l).C, S, \sigma, M, D\rangle$
배열 대입	$\langle \texttt{aassign}(l).C,$ $\langle l_1, n\rangle.\langle l_2, \langle a, o, s\rangle\rangle.\langle l_3, v\rangle.S, \sigma, M, D\rangle$	$\leadsto$	$\langle C, \langle l, v\rangle.S, \sigma, M[v/a+o+n], D\rangle$

대입(assignment)은 한 주소가 가리키는 메모리 위치(lvalue)에 새로운 값(rvalue)을 저장하는 것이다. 대입문 자체가 만드는 값은 메모리에 새롭게 저장되는 값이 되고 이 값이 데이터 스택에 쌓인다. 변수에 값을 대입하는 경우에는 환경에서 주소를 찾아오고, 포인터가 가리키는 메모리 주소에 값을 저장하는 경우에는 포인터 연산을 통해 주소를 계산한다.

제어

순서	$\langle e_1; e_2.C, S, \sigma, M, D\rangle$	$\leadsto$	$\langle e_1.e_2.C, S, \sigma, M, D\rangle$
함수 호출	$\langle (e_1\texttt{(}e_2\texttt{)})_l.C, S, \sigma, M, D\rangle$	$\leadsto$	$\langle e_2.e_1.\texttt{app}(l).C, S, \sigma, M, D\rangle$
함수 적용	$\langle \texttt{app}(l).C, \langle l', \langle f\texttt{(}x\texttt{)=}e, \sigma'\rangle\rangle.\langle l'', v\rangle.S,$ $\sigma, M, D\rangle$	$\leadsto$	$\langle e_{l''}, S, \sigma'[a/f, a'/x],$ $M[\langle f\texttt{(}x\texttt{)=}e, \sigma'\rangle/a, v/a'], \langle l, C, \sigma\rangle.D\rangle$ $\texttt{allocated}(l'') = \texttt{allocated}(l'') \cup \{a, a'\}$
반환문	$\langle \texttt{return}\ e.C, S, \sigma, M, D\rangle$	$\leadsto$	$\langle e.\texttt{ret}.C, S, \sigma, M, D\rangle$
반환	$\langle \texttt{ret}.C, \langle l, v\rangle.S, \sigma, M, \langle l', C', \sigma'\rangle.D\rangle$	$\leadsto$	$\langle C', \langle l', v\rangle.S, \sigma', M, D\rangle$
조건문	$\langle \texttt{if}e_1e_2e_3.C, S, \sigma, M, D\rangle$	$\leadsto$	$\langle e_1.\texttt{cond}(e_2, e_3).C, S, \sigma, M, D\rangle$
조건	$\langle \texttt{cond}(e_1, e_2).C, v.S, \sigma, M, D\rangle$	$\leadsto$	$\langle e'.C, S, \sigma, M, D\rangle$ If $v \neq 0,\ e' = e_2$. Otherwise, $e' = e_1$.
반복문	$\langle \texttt{while}\ e_1\ e_2.C, S, \sigma, M, D\rangle$	$\leadsto$	$\langle e_1.\texttt{loop}(e_1, e_2).C, S, \sigma, M, D\rangle$
반복	$\langle \texttt{loop}(e_1, e_2).C, \langle l, v\rangle.S, \sigma, M, D\rangle$	$\leadsto$	$\langle e_2.e_1.\texttt{loop}(e_1, e_2).C, S, \sigma, M, D\rangle$ $v \neq 0$
	$\langle \texttt{loop}(e_1, e_2).C, \langle l, 0\rangle.S, \sigma, M, D\rangle$	$\leadsto$	$\langle C, S, \sigma, M, D\rangle$

C'은 함수 호출, 분기, 반복 등에 필요한 프로그램 구성 요소들을 제공한다. 함수 호출은 함수 호출 식의 레이블(label)과 함수 호출 후에 해야할 명령문, 함수 호출 시의 환경을 제어 스택 D에 쌓고, 함수의 몸뚱이(body)를 다음에 할 일로 가져온다. 함수 호출 시에는 함수의 인자 값들을 위한 메모리 영역이 할당된다.이때 사용되는 주소는 모두 새로운 값이다. 반환문

$$
\begin{array}{rrcl}
v \in & \widehat{Val} & = & \widehat{\mathbb{Z}} \times \widehat{Closure} \times \widehat{Blk} \\
I \in & \widehat{\mathbb{Z}} & = & \{[a,b] | a,b \in \mathbb{Z} \cup \{-\infty,\infty\}, a \leq b\}_\perp \\
 & \widehat{Blk} & = & 2^{(Lab \times \widehat{\mathbb{Z}} \times \widehat{\mathbb{Z}})} \\
 & \widehat{Closure} & = & 2^{Fexpr} \\
S \in & \widehat{Stack} & = & Lab \xrightarrow{\text{fin}} \widehat{Val} \\
M \in & \widehat{Mem} & = & Lab \xrightarrow{\text{fin}} \widehat{Val} \\
D \in & \widehat{Dump} & = & Lab \xrightarrow{\text{fin}} 2^C \\
\langle C,S,M,D \rangle \in & \widehat{Machine} & = & C \times \widehat{Stack} \times \widehat{Mem} \times \widehat{Dump} \\
 & \widehat{State} & = & \widehat{Stack} \times \widehat{Mem} \times \widehat{Dump}
\end{array}
$$

Figure 3: 요약 의미 공간

은 반환 값을 데이터 스택에 쌓고, 제어 스택의 맨 위에 있는 명령문을 다음에 할 일로 가져온다. 이 때 환경은 제어 스택에 쌓아 두었던 것으로 복구한다.

2.3 요약 실행 의미

요약 상태 $\widehat{State}$는 요약 데이타 스택(abstract data stack) $\widehat{Stack}$, 요약 메모리(abstract memory) $\widehat{Mem}$, 요약 제어 스택(abstract dump) $\widehat{Dump}$로 구성되어 있다. $\widehat{Stack}$과 $\widehat{Mem}$은 모두 레이블에 값($\widehat{Val}$)을 대응시키는 맵(map)으로 정의되지만 각각의 의미는 다르다. $\widehat{Stack}$은 프로그램의 각 지점에 있는 식(expression)들이 만들어 내는 값을 담기위해 각 프로그램 식에 달린 레이블에 값을 대응시키는 맵이되고, $\widehat{Mem}$은 메모리 주소가 레이블(메모리 할당식에 달린 레이블 또는 변수 이름)로 요약되기 때문에 레이블에 값을 대응시키는 맵이된다.

요약 공간(abstract domain) 상에서 메모리 주소는 메모리가 할당된 프로그램 코드 상의 위치(allocation site) 또는 변수 이름으로 요약된다. 배열의 크기 및 오프셋은 모두 구간 도메인(interval domain) $\widehat{\mathbb{Z}}$ 상의 값으로 요약된다. 구간 도메인은 최소값과 최대값의 구간으로 표현 된다. 만약 어떤 변수 x가 특정 프로그램 포인트에서 가질 수 있는 값들이 $\{2,3,6\}$과 같다면 이를 포섭하는 구간 도메인 상의 안전한 요약은 $[2,6]$이 된다. 다음의 코드를 보자:

```
int p[5];
int *q = p + 3;
*(q+3) = 1;
```

크기가 5인 배열을 가리키는 변수 p의 요약 공간 상에서의 값은 $\langle l, [0,0], [5,5] \rangle$이 된다. 여기서 l은 배열의 요약 주소이다. $[0,0]$과 $[5,5]$는 각각 배열의 오프셋과 크기를 나타낸다. 포인터 q는 p+3으로 초기화 되어 $\langle l, [3,3], [5,5] \rangle$의 값을 가진다. 따라서 *(q+3)는 $\langle l, [6,6], [5,5] \rangle$를 가리키므로 배열 참조 오류(buffer overrun)가 된다는 것을 알 수 있다.

아이락의 분석은 프로그램의 실제 실행 의미를 프로그램 지점(program point) 별로 요약하는 것이다.따라서 프로그램 분석은 프로그램의 실제 실행시 일어날 수 있는 모든 상태를 프로그램의 각 지점 $Edge$에서 요약 상태 $\widehat{State}$로 가는 맵 $T \in Edge \to \widehat{State}$로 안전하게 요약하

는 것이다. T는 다음 함수$\widehat{\mathcal{F}}$의 최소 고정점(least fixpoint) 계산으로 구할 수 있다:

$$\widehat{\mathcal{F}} : (Edge \to \widehat{State}) \to (Edge \to \widehat{State})$$
$$\widehat{\mathcal{F}}(T) = \lambda\langle l, l'\rangle.s \quad \text{where } \langle c_l, \bigsqcup\{s' \mid p \in pred(l), T\langle p, l\rangle = s'\}\rangle \leadsto^{\#} \langle c_{l'}, s\rangle$$

위 함수에서 $\leadsto^{\#}$(2.5 절에서 정의)는 요약 기계 상태 전이를 나타내며, $pred(l)$는 레이블 l이 달린 프로그램 식 직전에 실행되는 프로그램 식들에 달린 레이블들의 집합을 의미한다.

2.4 요약 함수

아이락의 분석은 요약 의미 공간(abstract semantic domain) 상에서 정의된 의미 함수(semantic function)의 고정점(fixpoint)을 계산하는 것이다. 이렇게 얻은 고정점은 프로그램의 요약 의미이다. 이 요약 의미가 프로그램의 실제 실행을 포섭하기 위해서는 실제 의미 공간과 요약 의미 공간 상에 갈로아 연결(Galois connection)[5]이 존재해야 한다. 갈로아 연결은 요약 함수(abstraction)와 구체화 함수(concretization)의 짝으로 이루어지며, 둘 중 하나만 정의하면 다른 하나는 결정될 수 있으므로 여기서는 아이락의 요약 함수 α를 살펴 보도록 한다. 먼저 의미 공간이 CPO(complete partial order)가 되어야 갈로아 연결을 만들 수 있으므로, 실제 기계 상태 전이열을 멱집합으로 한 단수 높인다. 요약 함수 α를 다음과 같이 정의할 수 있다:

$$\begin{aligned} 2^{(Edge \times State)^*} &\overset{\alpha_1}{\to} Edge \to 2^{Stack} \times 2^{Env} \times 2^{Mem} \times 2^{Dump} \\ &\overset{\alpha_2}{\to} Edge \to \widehat{Stack} \times \widehat{Mem} \times \widehat{Dump} \\ &= Edge \to \widehat{State} \\ \alpha &= \alpha_2 \circ \alpha_1 \end{aligned}$$

요약 함수 α_1은 기계 상태 전이 열들을 프로그램 포인트(program point) 별로 뭉쳐(partitioning) 실행 순서를 요약한다. 이 때 환경은 제거한다. 환경은 프로그램 실행의 문맥(context)를 구분하기 위한 것인데, 아이락은 이런 환경을 제거함으로써 기본적으로 문맥 구분을 하지 않는 분석을 하게된다[4]. α_1의 정의는 다음과 같다:

$$\alpha_1\ X =$$
$$\lambda\langle l, l'\rangle.\langle\{s \mid \cdots\langle\langle l, l'\rangle, s, \_, \_\rangle\cdots \in X\}, \{m \mid \cdots\langle\langle l, l'\rangle, \_, m, \_\rangle\cdots \in X\}, \{d \mid \cdots\langle\langle l, l'\rangle, \_, \_, d\rangle\cdots \in X\}\rangle$$

아이락이 최종적으로 계산하는 프로그램의 요약 의미는 프로그램의 각 지점들에서 실행 중에 발생할 수 있는 모든 상태를 포섭하는 요약 상태이다. 이는 각 간선을 요약 상태로 대응 시키는 맵(map) $T \in Edge \to \widehat{State}$로 표현된다. 요약 함수 α_2는 그림 4의 α_{Stack}, α_{Mem}, α_{Dump}들로 정의할 수 있다:

$$\alpha_2\ X = \lambda\langle l, l'\rangle.\langle\alpha_{Stack}X\langle l, l'\rangle.1, \alpha_{Mem}X\langle l, l'\rangle.3, \alpha_{Dump}X\langle l, l\rangle.4\rangle$$

여기서 $S.i$는 S의 i번째 구성 요소를 의미한다.

α_{Stack}과 α_{Mem}은 각각 프로그램 식과 요약 주소 별로 모은 값들을 요약(α_{Val})한다. 프로그램 식과 요약 주소가 모두 레이블(label)로 표현되기 때문이다. α_{Val}은 $\widehat{\mathbb{Z}}$ 및 $\widehat{Blk}$의 합치기(join) 연산을 이용해 값들을 요약한다. α_{Dump}는 레이블별로 다음에 할일들을 모두 모은다. α_{Dump}는 다음에 할일을 모으는 것 자체가 요약이 된다.

[4] 아이락은 함수 펼치기(function inlining)을 통해 문맥 구분 분석을 할 수도 있다.

$$
\begin{array}{rcl}
\alpha_{\mathbb{Z}} : 2^{\mathbb{Z}} \to \widehat{\mathbb{Z}} & = & \lambda X.\text{if } X = \varnothing \text{ then } \bot \text{ else } [\min(X), \max(X)] \\
\alpha_{Closure} : 2^{Closure} \to \widehat{Closure} & = & \lambda X.\text{if } X = \varnothing \text{ then } \varnothing \text{ else } \{f(x)\texttt{=}e \mid \langle f(x)\texttt{=}e, \sigma\rangle \in X\} \\
\alpha'_{Blk} : Blk \to \widehat{Blk} & = & \lambda\langle a, n_1, n_2\rangle.\{\langle l, [n_1, n_1], [n_2, n_2]\rangle\} \\
 & & \text{where } a \in \texttt{allocated}(l) \\
\alpha_{Blk} : 2^{Blk} \to \widehat{Blk} & = & \lambda X.\text{if } X = \varnothing \text{ then } \bot \text{ else } \bigsqcup_{\widehat{Blk}} \{\alpha'_{Blk}(b) \mid b \in X\} \\
\alpha_{Val} : 2^{Val} \to \widehat{Val} & = & \lambda X.\langle \alpha_{\mathbb{Z}}(X.\mathbb{Z}), \alpha_{Closure}(X.Closure), \alpha_{Blk}(X.Blk)\rangle \\
 & & \text{where } X.S = X \cap S \\
\alpha_{Mem} : 2^{Mem} \to \widehat{Mem} & = & \lambda X.\lambda l.\alpha_{Val}(\bigcup_{M \in X}\{M(a) \mid a \in \texttt{allocated}(l)\}) \\
\alpha_{Stack} : 2^{Stack} \to \widehat{Stack} & = & \lambda X.\lambda l.\alpha_{Val}(\bigcup_{S \in X} \text{collect}_{Stack}(l, S)) \\
\alpha_{Dump} : 2^{Dump} \to \widehat{Dump} & = & \lambda X.\lambda l.\bigcup_{D \in X} \text{collect}_{Dump}(l, d) \\
\\
\text{collect}_{Stack} : Lab \times Stack \to 2^{Val} & = & \lambda(l, S).\{v \mid \langle l, v\rangle \in s\} \\
\text{collect}_{Dump} : Lab \times Dump \to 2^{C} & = & \lambda(l, D).\{c \mid \langle l', c, \sigma\rangle \in D, l = l'\} \\
\\
A \sqcup_{\widehat{Blk}} B & = & S\cup \\
 & & (A - \{\langle l, n, m\rangle \in A \mid \langle l, \_, \_\rangle \in S\})\cup \\
 & & (B - \{\langle l, n, m\rangle \in B \mid \langle l, \_, \_\rangle \in S\}) \\
S & = & \{\langle l, n \sqcup_{\mathbb{Z}} n', m \sqcup_{\mathbb{Z}} m'\rangle \mid \langle l, n, m\rangle \in A, \langle l, n', m'\rangle \in B\} \\
\top_{\mathbb{Z}} \sqcup_{\mathbb{Z}} \_ & = & \top_{\mathbb{Z}} \\
\_ \sqcup_{\mathbb{Z}} \top_{\mathbb{Z}} & = & \top_{\mathbb{Z}} \\
\bot_{\mathbb{Z}} \sqcup_{\mathbb{Z}} \_ & = & \bot_{\mathbb{Z}} \\
\_ \sqcup_{\mathbb{Z}} \bot_{\mathbb{Z}} & = & \bot_{\mathbb{Z}} \\
[a, b] \sqcup_{\mathbb{Z}} [c, d] & = & [\min\{a, c\}, \max\{b, d\}]
\end{array}
$$

Figure 4: $\mathbb{Z}$, *Closure*, *Blk*, *Val*, *Mem*, *Stack*, *Dump*의 요약 함수

2.5 요약 기계 상태 전이 규칙

요약 상태 전이 $\leadsto^{\#}$는 다음의 규칙으로 정의된다:

$$\langle c, S, M, D\rangle \leadsto^{\#} \{\langle c', S', M', D'\rangle \mid c'\text{는 } c \text{ 다음에 할 일}\}$$

여기서 $c, c' \in C$, $S, S' \in \widehat{Stack}$, $M, M' \in \widehat{Mem}$, $D, D' \in \widehat{Dump}$이다. 주어진 상태 $\langle S, M, D\rangle$를 가지고 명령문 c를 실행하면 다음에 할 명령문 c'들과 새로운 상태 $\langle S', M', D'\rangle$가 결정된다. 요약 실행에서는 분기문의 조건식이 동시에 참도 되고 거짓도 될 수 있으므로 한 명령 수행 후 해야할 일이 둘 이상일 수 있다. 또한 각 지점의 함수 호출을 함수 별로 뭉치므로(context insensitivity) 함수 반환 이후에 할 일도 둘 이상일 수가 있다. 이하의 요약 기계 상태 전이 규칙 정의에서는 다음에 할 명령문이 둘 이상인 경우에만 명확하게 집합으로 표시하고, 다음에

할 명령문이 하나인 경우에는 집합으로 표시하지 않는다.

명령문 경우별 $\hookrightarrow^{\#}$의 정의를 보기 전에 $\hookrightarrow^{\#}$를 정의하는데 사용한 연산들의 정의를 살펴보자:

- 함수 갱신

$$\begin{array}{rcl} f[y/\!\!/x] & = & \begin{cases} f[y/x] & \text{if } x \notin dom f \\ f[f(x) \sqcup y/x] & \text{otherwise} \end{cases} \\ f[y/\!\!/\{x_1, \ldots, x_n\}] & = & f[y/\!\!/x_1, \ldots, y/\!\!/x_n]. \end{array}$$

- $v \in \widehat{Val}$일 때 $v.i$는 v의 i번째 구성 요소를 취한 것이다.
- $v \in \widehat{Val}$일 때 $v.X$는 v의 구성 요소 중 X형을 취한 것이다.
- $v \in \widehat{Blk}$일 때 $v.1 = \{l \mid \langle l, m, n\rangle \in v\}$이다.

2.5.1 선언

변수	$\langle x.C, S, M, D\rangle$	$\hookrightarrow^{\#}$	$\langle C, S, M[\bot/x], D\rangle$
배열	$\langle x\texttt{[}n\texttt{]}_l.C, S, M, D\rangle$	$\hookrightarrow^{\#}$	$\langle C, S, M[\langle l, [0,0], [n,n]\rangle/x, \bot/l], D\rangle$
함수	$\langle (f\texttt{(}x\texttt{)=}e).C, S, M, D\rangle$	$\hookrightarrow^{\#}$	$\langle C, S, M[\{f\texttt{(}x\texttt{)=}e\}/f], D\rangle$

요약 의미 공간의 주소는 메모리가 할당된 위치 및 변수 이름으로 요약 된다. 요약 실행 의미에서는 배열의 경우 각 원소에 대한 메모리는 따로 할당되지 않고 하나의 메모리 영역에 배열 원소의 모든 값이 합쳐지므로(join) 배열 원소 전체에 대한 메모리는 할당되지 않는다.

2.5.2 프로그램 식

메모리 할당

배열	$\langle \texttt{[}e_{l'}\texttt{]}_l.C, S, M, D\rangle$	$\hookrightarrow^{\#}$	$\langle e_{l'}.\texttt{allocar}(l, l').C, S, M, D\rangle$
배열 할당	$\langle \texttt{allocar}(l, l').C, S, M, D\rangle$	$\hookrightarrow^{\#}$	$\langle C, S[\langle l, [0,0], S(l')\rangle/\!\!/l], M[\bot/\!\!/l], D\rangle$

동적으로 할당되는 배열은 메모리 할당식의 레이블(allocation site)을 시작 주소로 가진다. 배열 할당 식의 값은 배열의 시작 주소, 오프셋(offset), 크기로 이루어진 $\widehat{Blk}$ 값이 된다. 배열 선언과 마찬가지로 배열 원소 전체에 대한 메모리 할당은 일어나지 않는다.

값

상수	$\langle v_l.C, S, M, D\rangle$	$\hookrightarrow^{\#}$	$\langle C, S[v_1/\!\!/l], M, D\rangle$
이항 연산	$\langle e_{l_1}\ \mathrm{bop}_l\ e_{l_2}.C, S, M, D\rangle$	$\hookrightarrow^{\#}$	$\langle e_{l_1}.e_{l_2}.\texttt{bop}(l, l_1, l_2).C, S, M, D\rangle$
	$\langle \texttt{bop}(l, l_1, l_2).C, S, M, D\rangle$	$\hookrightarrow^{\#}$	$\langle C, S[S(l_1)\ \hat{\mathrm{bop}}\ S(l_2)/\!\!/l], M, D\rangle$
포인터	$\langle (\texttt{*}e_{l'})_l.C, S, M, D\rangle$	$\hookrightarrow^{\#}$	$\langle e_{l'}.\texttt{deref}(l, l').C, S, M, D\rangle$
포인터 값	$\langle \texttt{deref}(l_1, l_2).C, S, M, D\rangle$	$\hookrightarrow^{\#}$	$\langle C, S[\bigsqcup_{\langle l,n,m\rangle \in S(l_2).\widehat{Blk}} M(l)/\!\!/l_1], M, D\rangle$
변수	$\langle \mathtt{x}_l.C, S, M, D\rangle$	$\hookrightarrow^{\#}$	$\langle C, S[M(x)/\!\!/l], M, D\rangle$
배열 원소	$\langle e_{l_1}\texttt{[}e_{l_2}\texttt{]}_l.C, S, M, D\rangle$	$\hookrightarrow^{\#}$	$\langle e_{l_1}.e_{l_2}.\texttt{aval}(l, l_1, l_2).C, S, M, D\rangle$
	$\langle \texttt{aval}(l, l_1, l_2).C, S, M, D\rangle$	$\hookrightarrow^{\#}$	$\langle C, S[\bigsqcup_{\langle l',n,m\rangle \in S(l_1).\widehat{Blk}} M(l')/\!\!/l], M, D\rangle$

값만 계산하는 프로그램 식은 메모리 변화 없이 프로그램 식의 값이 데이터 스택에 합쳐지기만 한다. 이항 연산 $\hat{\text{bop}}$는 요약 의미 공간 $\widehat{Val}$에서 정의된다. 예를 들어 요약 의미 공간의 포인터 연산이라 할 수 있는 $X \in \widehat{Blk}$와 $[m,n] \in \widehat{\mathbb{Z}}$의 합 $\hat{+}$는 다음과 같이 정의된다:

$$X \hat{+} [m,n] = \{\langle l, [o_1 + m, o_2 + n], [s_1, s_2]\rangle \mid \langle l, [o_1, o_2], [s_1, s_2]\rangle \in X\}$$

대입

변수	$\langle x \texttt{=}_{l_1} e_{l_2}.C, S, M, D\rangle$	$\leadsto^{\#}$	$\langle e_{l_2}.\texttt{vassign}(l_1, l_2, x).C, S, M, D\rangle$
변수 대입	$\langle \texttt{vassign}(l_1, l_2, x).C, S, M, D\rangle$	$\leadsto^{\#}$	$\langle C, S[S(l_2) /\!\!/ l_1], M[S(l_2) /\!\!/ x], D\rangle$
포인터	$\langle (*e_{l_1})_{l_3} \texttt{=}_l e_{l_2}.C, S, M, D\rangle$	$\leadsto^{\#}$	$\langle e_{l_2}.e_{l_1}.\texttt{assign}(l, l_2, l_1).C, S, M, D\rangle$
주소 대입	$\langle \texttt{assign}(l, l_1, l_2).C, S, M, D\rangle$	$\leadsto^{\#}$	$\langle C, S[S(l_1) /\!\!/ l], M[S(l_1) /\!\!/\!\!/ S(l_2).\widehat{Blk}.1], D\rangle$
배열	$\langle e_{l_1} \texttt{[}e_{l_2}\texttt{]=}_l e_{l_3}.C, S, M, D\rangle$	$\leadsto^{\#}$	$\langle e_{l_3}.e_{l_1}.e_{l_2}.\texttt{aassign}(l, l_3, l_1, l_2).C, S, M, D\rangle$
	$\langle \texttt{aassign}(l, l_1, l_2, l_3).C, S, M, D\rangle$	$\leadsto^{\#}$	$\langle C, S[S(l_1) /\!\!/ l], M[S(l_1) /\!\!/\!\!/ S(l_2).\widehat{Blk}.1], D\rangle$

대입문은 데이터 스택과 메모리를 모두 변화 시킨다. 메모리는 값이 저장될 주소에 새로운 값이 합쳐지고, 데이터 스택은 대입문에 해당하는 레이블(label)이 가리키는 값에 새로운 값이 합쳐짐으로써 갱신된다. 배열 원소에 값을 대입하는 경우에는 배열 인덱스(index)에 상관 없이 모든 배열 원소 값이 배열의 시작 주소가 가리키는 하나의 메모리 영역에 합쳐진다(join).

제어

순서	$\langle e_{l_1}; e_{l_2}.C, S, M, D\rangle$	$\hookrightarrow^{\#}$	$\langle e_{l_1}.e_{l_2}.C, S, M, D\rangle$
함수 호출	$\langle e_{l_1}(e_{l_2})_l.C, S, M, D\rangle$	$\hookrightarrow^{\#}$	$\langle e_{l_2}.e_{l_1}.\mathtt{app}_1(l, l_2, l_1).C, S, M, D\rangle$
함수 적용	$\langle \mathtt{app}_1(l, l_1, l_2).C, S, M, D\rangle$	$\hookrightarrow^{\#}$	$\{\langle \mathtt{app}_2(l, l_1, f\,\mathtt{(}x\mathtt{)=}e).C, M, S, D\rangle \mid f\,\mathtt{(}x\mathtt{)=}e \in S(l_2)\}$
	$\langle \mathtt{app}_2(l, l_p, f\,\mathtt{(}x\mathtt{)=}e_{l_f}).C, S, M, D\rangle$	$\hookrightarrow^{\#}$	$\langle e.\mathtt{return}^{l_f}\ \bot, S, M[S(l_p)/\!\!/x], D[\{\mathtt{fetch}(l_f, l).C\}/\!\!/l_f]\rangle$
꺼내오기	$\langle \mathtt{fetch}(l, l').C, S, M, D\rangle$	$\hookrightarrow^{\#}$	$\langle C, S[S(l)/\!\!/l'], M, D\rangle$
반환문	$\langle \mathtt{return}^{l_f}\ e_l.C, S, M, D\rangle$	$\hookrightarrow^{\#}$	$\langle e_l.\mathtt{ret}(l_f, l).C, S, M, D\rangle$
반환	$\langle \mathtt{ret}(l_f, l).C, S, M, D\rangle$	$\hookrightarrow^{\#}$	$\{\langle C, S[S(l)/\!\!/l_f], M, D\rangle \mid C \in D(l_f)\}$
조건문	$\langle \mathtt{if} e_l e_2 e_3.C, S, M, D\rangle$	$\hookrightarrow^{\#}$	$\langle e_l.\mathtt{cond}(l, e_2, e_3).C, S, M, D\rangle$
조건	$\langle \mathtt{cond}(l, e_1, e_2).C, S, M, D\rangle$	$\hookrightarrow^{\#}$	$\begin{cases} \langle \epsilon, \bot, \bot, \bot\rangle & \bot = S(l).\widehat{\mathbb{Z}} \\ \langle e_1.C, S, M, D\rangle & 0 \notin S(l).\widehat{\mathbb{Z}} \\ \langle e_2.C, S, M, D\rangle & [0,0] = S(l).\widehat{\mathbb{Z}} \\ \{\langle e_1.C, S_{(t,l)}, M_{(t,l)}, D\rangle, & \\ \langle e_2.C, S_{(f,l)}, M_{(f,l)}, D\rangle\} & \text{otherwise} \end{cases}$
반복문	$\langle \mathtt{while}\ e_{l_1}\ e_{l_2}.C, S, M, D\rangle$	$\hookrightarrow^{\#}$	$\langle e_{l_1}.\mathtt{loop}(l_1, e_{l_1}, e_{l_2}).C, S, M, D\rangle$
반복	$\langle \mathtt{loop}(l, e_1, e_2).C, S, M, D\rangle$	$\hookrightarrow^{\#}$	$\begin{cases} \langle \epsilon, \bot, \bot, \bot\rangle & \bot = S(l).\widehat{\mathbb{Z}} \\ \langle e_2.e_1.\mathtt{loop}(l, e_1, e_2).C, S, M, D\rangle & 0 \notin S(l).\widehat{\mathbb{Z}} \\ \langle C, S, M, D\rangle & [0,0] = S(l).\widehat{\mathbb{Z}} \\ \{\langle C, S_{(t,l)}, M_{(t,l)}, D\rangle, & \\ \langle e_2.e_1.\mathtt{loop}(l, e_1, e_2).C, S_{(f,l)}, M_{(f,l)}, D\rangle\} & \text{otherwise} \end{cases}$

실제 실행 의미에서 함수는 호출된 상황(context)에 따라 다른 상태를 가질 수 있지만, 요약 실행 의미에서 함수는 실제 실행의 모든 함수 호출들을 함수별로 뭉쳐서 분석된다. 함수의 형식 매개 변수(formal parameter)에 대입되는 모든 실제 매개 변수(actual parameter)는 각 변수 별로 합쳐지고, 함수가 반환하는 값은 함수별로 합쳐진다. 이를 위해 반환문(return)은 자신이 속한 함수를 가리키는 레이블(label)을 가지고 있다가 이 레이블을 이용해 반환값을 요약 데이터 스택에 합친다. 반환 후 수행해야할 명령들도 합쳐져서 요약 실행 의미에서는 함수 반환 후 실행해야하는 명령문이 둘 이상일 수 있다. 조건식이 있는 분기문이나 반복문에서는 가지치기(pruning)를 적용할 수 있다. 가지치기 연산은 그림 5에 정의되어 있다.

2.6 고정점 계산 알고리즘

고정점을 구하는 알고리즘은 '*할 일만 하기(working set)*' 알고리즘이다. 다음에 할 일 집합은 다시 계산해야하는 프로그램 식(expression)에 해당하는 레이블(label)의 집합이다. 프로그램 레이블(label) l의 프로그램 식을 다시 계산한 기계의 상태 $T\langle l, l'\rangle$가 변했다면 레이블(label) l'의 프로그램 식은 다시 계산해야 하므로 다음에 할 일 집합에 추가된다. 이 집합은 기본적으로 스택(stack)으로 구현되어 있다. 따라서, 할 일만 하기(working set) 알고리즘은 깊이 우선 탐색(depth-first traverse)으로 프로그램의 실행을 따르게 된다. 만약 다음에 계산해야 할 프

데이터 스택 가지치기

$$S_{(t,l)} = S\{l \mapsto [1,1]\}$$
$$S_{(f,l)} = S\{l \mapsto [0,0]\}$$

메모리 가지치기

- *case* $e_l : x < c$

$$\begin{cases} M_{(t,l)} = M\{x \mapsto M(x) \sqcap [-\infty, c-1]\} \\ M_{(f,l)} = M\{x \mapsto M(x) \sqcap [c, +\infty]\} \end{cases}$$

- *case* $e_l : x < c_1 y + c_2$

$$\begin{cases} M_{(t,l)} = M\{x \mapsto M(x) \sqcap [-\infty, \max_{\widehat{\mathbb{Z}}}(c_1 * M(y) + c_2 - 1)] \\ \qquad\qquad y \mapsto M(y) \sqcap [\min_{\widehat{\mathbb{Z}}}((M(x) - c_2)/c_1 + 1), +\infty]\} & \text{when } c_1 > 0 \\ M_{(t,l)} = M\{x \mapsto M(x) \sqcap [-\infty, \max_{\widehat{\mathbb{Z}}}(c_1 * M(y) + c_2 - 1)] \\ \qquad\qquad y \mapsto M(y) \sqcap [-\infty, \max_{\widehat{\mathbb{Z}}}((M(x) - c_2)/c_1)]\} & \text{when } c_1 < 0 \\ M_{(f,l)} = M\{x \mapsto M(x) \sqcap [\min_{\widehat{\mathbb{Z}}}(c_1 * M(y) + c_2), +\infty] \\ \qquad\qquad y \mapsto M(y) \sqcap [-\infty, \max_{\widehat{\mathbb{Z}}}((M(x) - c_2)/c_1)]\} & \text{when } c_1 > 0 \\ M_{(f,l)} = M\{x \mapsto M(x) \sqcap [\min_{\widehat{\mathbb{Z}}}(c_1 * M(y) + c_2), +\infty] \\ \qquad\qquad y \mapsto M(y) \sqcap [\min_{\widehat{\mathbb{Z}}}((M(x) - c_2)/c_1), +\infty]\} & \text{when } c_1 < 0 \end{cases}$$

- *case* $e_l : x < c_1 y + c_2 z$

$$\begin{cases} M_{(t,l)} = M\{x \mapsto M(x) \sqcap [-\infty, \max_{\widehat{\mathbb{Z}}}(c_1 * M(y) + c_2 * M(z) - 1)], \\ \qquad\qquad y \mapsto M(y) \sqcap [\min_{\widehat{\mathbb{Z}}}((M(x) - c_2 * M(z) + 1)/c_1), +\infty], & \text{when } c_1 > 0 \\ \qquad\qquad z \mapsto M(z) \sqcap [\min_{\widehat{\mathbb{Z}}}((M(x) - c_1 * M(y) + 1)/c_2), +\infty]\} & \text{when } c_2 > 0 \\ M_{(f,l)} = M\{x \mapsto M(x) \sqcap [\min_{\widehat{\mathbb{Z}}}(c_1 * M(y) + c_2 * M(z)), +\infty]\} \\ \qquad\qquad y \mapsto M(y) \sqcap [-\infty, \max_{\widehat{\mathbb{Z}}}((M(x) - c_2 * M(z))/c_1)], & \text{when } c_1 > 0 \\ \qquad\qquad z \mapsto M(z) \sqcap [-\infty, \max_{\widehat{\mathbb{Z}}}((M(x) - c_1 * M(y))/c_2)]\} & \text{when } c_2 > 0 \end{cases}$$

$\widehat{\mathbb{Z}}$의 최대, 최소

- $\min_{\widehat{\mathbb{Z}}} = \lambda X.\min\{n1 \mid [n1, n2] \in X.\widehat{\mathbb{Z}}\}$
- $\max_{\widehat{\mathbb{Z}}} = \lambda X.\max\{n2 \mid [n1, n2] \in X.\widehat{\mathbb{Z}}\}$

Figure 5: 가지치기(pruning) 연산

로그램 식의 수가 여러 개이면(`if`문이나 `while`문에서 조건식이 참도 되고 거짓도 될 때) 다음에 계산해야 할 것들을 하나로 묶어 다음에 할 일 집합에 넣는다. 이렇게 함으로써 `if`문에서는 넓이 우선 탐색(breadth-first traverse)을 `while`문에서는 루프(loop)의 몸뚱이(body)를 먼저 계산하도록 하는 효과를 얻는다.

```
Input:  c_{l_0} : C                    (* a C' program *)
Output: T : (Lab × Lab) → State^

proc Tabulate(F^ : (Lab × State^) → 2^{(Lab × State^)}, c_{l_o} : C)
    W : (2^{Lab})*          (* workset, statck of label bags *)
    B, B' : 2^{Lab}         (* bag of labels to re-evaluate *)
    G : 2^{(Lab × Lab)}     (* flow graph edges *)
    l : Lab,        y : State^
    proc Iterator(⊙ : State^ × State^ → State^, ⊑: State^ × State^ → {true, false})
        W := push(ε, {l_0})
        while W ≠ ∅
            (W, B') := pop(W)
            B := ∅
            foreach l ∈ B'
                foreach (n, y) ∈ F^(l, ⨆_{(p,l)∈G} T(p, l))
                    G := G ∪ {(l, n)}
                    if widen(l, G) then
                        y := T(l, n) ⊙ y
                    if y ⋢ T(l, n) then
                        B := B ∪ {n}
                        T(l, n) := y
                end
            end
            W := push(W, B)
        end
    T := λ(l, l').⊥_{State^}
    G := ∅
    (* widening iterations *)
    Iterator(▽, ⊑)
    (* narrowing iterations *)
    Iterator(△, ⊒)
```

Figure 6: 고정점 계산 알고리즘 *Tabulate*

고정점을 구하는 알고리즘은 두 부분으로 구성되어 있다. 우선 축지법(widening)을 사용해 주어진 프로그램을 분석한다. 축지법 분석 후 좁히기(narrowing)를 사용한 분석을 수행한다. 축지법(widening)은 정수 구간 공간의 높이가 무한하기 때문에 유한한 시간에 분석을 끝내기 위해서 필요하다. 축지법(widening)을 사용하면 최소 고정점 큰 값에 도달하게 되어 분석 정확도가 떨어지게 된다. 좁히기(narrowing)는 이렇게 떨어진 정확도를 어느 정도 복

구 시켜준다. 아이락은 축지법(widening)을 하는 곳과 좁히기(narrowing)를 하는 곳을 효율적으로 결정한다. 예를 들어 프로그램 흐름 그래프(control flow graph)의 루프 진입 지점(loop head)에서만 축지법(widening)을 한다.

요약 기계 상태 전의 정의로 부터 그림 6의 함수 $\widehat{F}$는 다음과 같이 정의할 수 있다:

$$\begin{array}{rcl} \widehat{F} & : & Lab \times \widehat{State} \to 2^{Lab \times \widehat{State}} \\ \widehat{F}\langle l, x\rangle & = & \{\langle l', x'\rangle \mid \langle l, x\rangle \leadsto^{\#} X,\ \langle l', x'\rangle \in X\} \end{array}$$

그림 6의 $widen(l, G)$는 프로그램 포인트 l이 프로그램의 흐름 그래프에서 축지법 포인트에 속할 경우에 참 값을 갖고 그렇지 않을 경우는 거짓 값을 갖는다. 그림 6의 알고리즘 $Tabulate$는 주어진 프로그램 c_{l_o}에 대해 프로그램의 각 지점의 요약 상태 $T \in Edge \to \widehat{State}$를 계산해 낸다.

2.7 정확도 향상을 위한 기술

아이락은 분석의 정확도 향상을 위하여 프로그램 분석 분야에서 오랫동안 축적해온 여러 기술들을 사용한다. 각 기술들은 서로 독립적이기 때문에 모든 기술들을 조합하여 사용하면 각각을 사용하였을 때에 비해 더 정확한 분석 결과를 얻을 수 있다

- 새롭게 이름짓기(Unique Renaming) 요약 의미 공간에는 환경(environment)이 존재하지 않는다. 요약 주소는 식별자 이름과 메모리 할당 위치(allocation site)로 정의되므로, 환경이 없는 상태에서 같은 이름을 가진 식별자가 많으면 분석의 정확도가 떨어질 수 있다. 이를 피하기 위해 모든 식별자들은 각각 고유한 이름으로 다시 매겨진다.

- 축지법 후에 좁히기(Narrowing After Widening[7]) 아이락의 요약 공간(abstract domain)을 구성하는 요소 중 유일하게 정수구간 공간(interval domain)에서만 길이가 무한 체인(chain)을 정의할 수 있다. 축지법(Widening) 연산자는 무한한 공간에서의 계산을 유한한 시간 안에 끝내기 위해 필수적으로 사용해야한다. 축지법(widening) 연산을 사용하여 구한 고정점(fixpoint)은 최소고정점(least fixpoint)보다 크다. 따라서, 분석의 정확도가 상대적으로 떨어진다. 정확도의 복구를 위해 이 계산된 고정점으로부터 최소고정점에 근접한 고정점을 구하는 과정이 필요하다. 이를 위해 좁히기(Narrowing) 연산이 사용된다. 아이락은 배열의 인덱스(index)에 특화된 축지법(widening)과 좁히기(narrowing) 연산을 사용한다. 정수구간 공간 상에서의 축지법(widening)과 좁히기(narrowing) 연산자는 다음과 같이 정의된다:

  – 축지법(widening)

$$\begin{array}{rcl} \bot \triangledown X & = & X \\ X \triangledown \bot & = & X \\ [l_0, u_0] \triangledown [l_1, u_1] & = & [0 \le l_1 < l_0\ ?\ 0 : (l_1 < l_0\ ?\ -\infty : l_0), \\ & & \ 0 \ge u_1 > u_0\ ?\ 0 : (u_0 < u_1\ ?\ +\infty : u_0)] \end{array}$$

– 좁히기(narrowing)

$$\begin{aligned} \bot \vartriangle X &= \bot \\ X \vartriangle \bot &= \bot \\ [l_0, u_0] \vartriangle [l_1, u_1] &= [((l_0 \leq 0 \leq l_1) \vee (l_0 = -\infty)) \; ? \; l_1 : l_0, \\ &\quad\; ((u_1 \leq 0 \leq u_0) \vee (u_0 = +\infty)) \; ? \; u_1 : u_0] \end{aligned}$$

- 실행 순서 고려 분석(Flow Sensitive Analysis) 아이락은 프로그램 포인트(program point) 별로 실행 중 일어날 수 있는 모든 상태를 요약해 낸다. 즉 실행 순서를 고려하여 분석함으로써 높은 분석 정확도를 유지한다. 이를 위해 메모리 덮어쓰기(destructive update)를 지원 한다.

- 가지 치기(Context Pruning) 분기문이나 반복문의 조건식(conditional expression)을 참이 되게 하는 혹은 거짓이 되게 하는 최소의 기계 상태를 이용해 분석의 정확도를 높인다. 예를 들어 다음의 코드를 보자:

```
int A[10];
while (i<10) {
   A[i] = 1;
}
```

  위의 프로그램에서 반복문 내의 시작 지점에서는 i<10, 반복문 밖의 시작 지점에서는 i≥10 이 항상 유지된다는 사실을 알 수 있다. 이를 이용해 i 값을 보다 정확한 값으로 추정할 수 있다. 아이락은 변수에 대한 선형수식(linear expression)에 대해서는 이러한 가지치기를 지원한다. 가지치기는 소위 역방향 분석(backward analysis)이라고도 한다.

- 함수 펼치기(Function Inlining) 분석 중에 함수 몸뚱이(doby)를 펼친다. 함수 펼치기를 통해 한 함수가 여러 번 호출 되더라도 각 함수의 상태를 호출 지점 별로 계산해 낼 수 있다. 그런데 만약 무한히 재귀호출되는 함수에서도 펼치기를 계속하게 되면 분석이 끝나지 않을 수가 있다. 즉 고정점에 도달하지 않을 수 있다. 이를 막기 위해 사용자가 정해준 회수 이상으로는 함수 펼치기를 하지 않는다.

- 루프 펼치기(Loop Unrolling) 분석 중에 루프(loop)를 펼친다. 함수 펼치기와 마찬가지로 프로그램 분석이 유한한 시간 안에 끝나도록 사용자가 정해준 회수 이상으로 루프(loop)를 펼치지는 않는다.

- 오류 상황 후 분석(Error Recovery) C 언어에는 오류 상황이 발생했을 때의 실행 의미가 정해져 있지 않다. 배열 참조 오류같은 오류 상황 후에도 분석을 계속 진행하기 위해 우리는 낙관적인 가정을 한다. 다음의 예를 보자:

```
1: int A[10],j;
2: for (i=0; i<10; i++) {
3:    A[i] = 2 * i;
4: }
5: j = A[i];
6: ...A[i] ...
```

위의 예에서 5번 줄에서 i의 값은 $[10, 10]$이 된다. A는 크기가 10인 배열이기 때문에 올바른 인덱스(index)는 [0, 9]에 속해야 한다. i의 값이 이를 벗어나므로 배열 참조 오류이다. 그런데, 그 후에 i가 A의 인덱스(index)로 쓰이는 경우는 항상 경보를 발생 시키는 문제가 있다. 또, j의 값을 무엇으로 할 지 정해야 하는 문제가 있다. 우리는 여기서 프로그래머가 배열 A의 인덱스(index)로 사용하는 i가 $[0, 9]$에 속하길 의도 했다는 것과 j의 값은 배열 A의 원소가 가질 수 있는 값을 의도했다는 것을 가정한다. 따라서 5번 줄 이후에서 i의 값은 $[0, 9]$, j의 값은 $[0, 18]$이라는 가정하에 분석을 진행한다. 이를 통해 앞에서 오류의 의해 연쇄적으로 일어나는 경보들을 피할 수 있다. 이는 사용자의 편의와 분석의 정확도를 위한 우리의 선택이다.

2.8 분석 비용을 줄이는 기술

분석 비용을 줄이기위해 아이락은 다음의 기술들을 사용하였다.

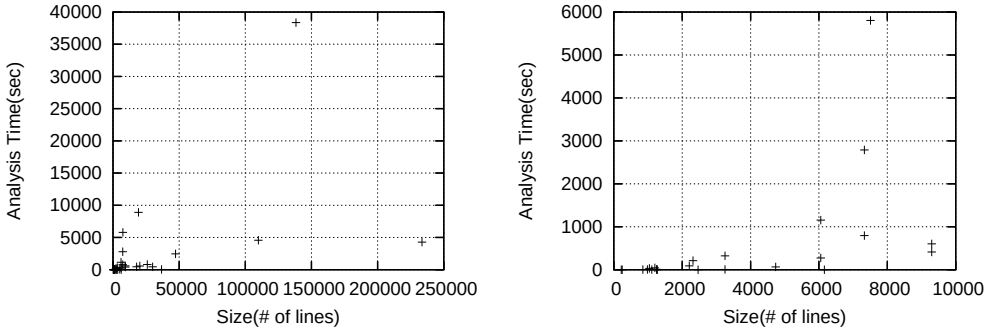
- 선택적 합치기/비교하기 연산(Selective Join/Order) 합치기/비교 연산에 소요되는 시간은 분석 시간의 대부분을 차지하기 때문에, 이 연산에 소요되는 시간을 줄이면 분석시간을 많은 부분 줄일 수 있다. 합치기/비교 연산의 대상이 되는 상태의 모든 원소들에 대해서 연산을 적용하는 대신에 그 원소중에 어떤 원소가 변화했는지에 대한 정보를 가지고, 변화된 원소들에 대해서만 합치기/비교 연산을 취한다.

- 분석 지연 시키기(Wait-at-Join) 분석 중에 여러 실행흐름들이 모이는 곳은 모여든 실행의 분석이 끝날 때까지 기다린 후에 분석을 실행한다. 이를 통해 분석 중의 중복 계산을 줄일 수 있다. 이 방법은 분기문의 양쪽이 모이는 곳이나 레이블 식(labeled expression)처럼 여러 실행 흐름이 모이는 접합점에 대한 계산을 대해 효과적으로 할 수 있게 한다. 아이락은 모든 접합점에 대한 계산은 무조건 연기하고 접합점이 아닌 곳에 대한 계산을 먼저 수행한다. 그러다가 접합점이 아닌 곳에 대한 할 일이 없어지면 기다리던 지점들부터 다시 분석하는 방법을 택하였다.

- 데이터 스택 제거(Stack Obviation) 프로그램의 요약 상태를 구성 요소 중 하나인 요약 데이타 스택(abstract data stack)은 다른 구성 요소인 요약 메모리(abstract memory)나 요약 제어 스택(abstract dump) 비해 상당히 크다. 이는 데이터 스택이 프로그램 중간에 발생하는 모든 임시 값들을 기억하는 데 사용되고, 따라서 그 크기가 프로그램의 크기에 비례하기 때문이다. 자연히 분석 시간의 대부분은 데이터 스택을 합치고 비교하는 데 소요된다. 만약 데이터 스택의 변화가 모두 메모리에 반영 되기만 한다면 데이터 스택은 합치거나 비교할 때 쳐다보지 않아도 된다. 아이락의 요약 실행 의미에서 이 조건이 성립되지 않는 경우는 C 언어의 삼항 연산자 ?:과 함수의 리턴 값을 기억해야하는 경우 밖에는 없다. 아이락은 이 두가지 경우에 대해 적절한 변환을 해줌으로써 데이터 스택의 모든 변화가 메모리에 반영되게 한다.

```
y = (x > 0) ? 1 : 2;
```

위 의 예에서 x의 값이 $[-1, 1]$이라고 하면 x는 0보다 작다고도 크다고도 말할 수 있다. 따라서 y의 값은 안전한 요약인 $[1, 2]$가 되어야한다. 그런데, 데이터 스택 합치기(join)를



(a) 프로그램의 크기와 분석 시간 (b) (a)의 왼쪽 아래를 확대한 그림

Figure 7: 아이락 분석 속도의 확장성

하지 않는다면 프로그램식의 임시 값인 1과 2가 합쳐지지 않아서 y의 값을 안전하게 요약해낼 수 없다. 하지만 아래와 같이 코드를 변환하면 임시 변수 tmp에 의해 삼항연산식의 값이 메모리에 반영되므로 데이터 스택을 무시하고 메모리를 합치는 것만으로도 안전한 y의 값을 구할 수 있다.

```
if (x > 0) tmp=1; else tmp=2;
y=tmp;
```

함수의 리턴 값에서도 이와 비슷한 변환을 해서 데이터 스택의 모든 변화가 메모리에 반영 되도록 했다. 이는 상태의 구성 요소 중에 가장 큰 데이터 스택을 합치기/비교에서 하지 않아도 되게 하여 상당한 분석시간 감소를 가져왔다.

2.9 분석기의 성능

우리는 nML 프로그래밍 시스템[5]으로 아이락을 구현하여 다양한 소프트웨어들을 분석해 보았다. 아이락은 GNU 소프트웨어, 리눅스 커널(Linux kernel) 프로그램과 상업용 내장형 소프트웨어(embedded software)에서 여러 가지 배열 참조 오류(buffer overrun)들을 찾아내었다.

표1은 아이락을 이용한 분석 실험 결과를 보여준다. 모든 분석은 펜티엄4 3.2GHz CPU와 4GB 메모리를 가진 리눅스 커널 버전 2.6 시스템에서 이뤄졌다. 리눅스 커널 프로그램에 대한 분석 결과를 보면 허위 경보 비율이 크지 않음을 볼 수 있다. GNU 소프트웨어는 이미 디버깅(debugging)이 많이 이루어진 프로그램이어서 오류가 적은 것으로 보인다. 상업용 내장형 소프트웨어 중의 일부는 아직 개발 중인 단계였고, 어떤 것들은 이미 테스트 단계를 거친 것들이었다. 아이락은 테스트 단계를 거친 프로그램에 대해서도 오류를 찾아냈다.

그림7은 아이락의 확장성(scalability)을 보여준다. 그림7의 그래프에서 볼 수 있듯이 분석 시간이 꼭 프로그램의 크기에 비례하는 것은 아니다. 아이락은 실제 실행의 흐름을 따르기 때문에 실행의 복잡도가 분석속도에 보다 큰 영향을 미친다. 아이락의 분석 시간은 프로그램 크기를 n으로 했을 때 최악의 경우(worst case)에 $O(n^3)$이다.

[5]ML의 한국 사투리. `http://ropas.snu.ac.kr/n`

소프트웨어		줄 수	CPU 시간	경보 수		실제
				배열[a]	접근[b]	오류 수
GNU	tar-1.13	20,258	576.79s	24	66	1
	bison-1.875	25,907	809.35s	28	50	0
	sed-4.0.8	6,053	1154.32s	7	29	0
	gzip-1.2.4a	7,327	794.31s	9	17	0
	grep-2.5.1	9,297	603.58s	2	2	0
Linux kernel 2.6.4	vmax302.c	246	0.28s	1	1	1
	xfrm_user.c	1,201	45.07s	2	2	1
	usb-midi.c	2,206	91.32s	2	10	4
	atkbd.c	811	1.99s	2	2	2
	keyboard.c	1,256	3.36s	2	2	1
	af_inet.c	1,273	1.17s	1	1	1
	eata_pio.c	984	7.50s	3	3	1
	cdc-acm.c	849	3.98s	1	3	3
	ip6_output.c	1,110	1.53s	0	0	0
	mptbase.c	6,158	0.79s	1	1	1
	aty128fb.c	2,466	0.32s	1	1	1
Commercial embedded software	software 1	109,878	4525.02s	16	64	1
	software 2	17,885	463.60s	8	18	9
	software 3	3,254	5.94s	17	57	0
	software 4	29,972	457.38s	10	140	112
	software 5	19,263	8912.86s	7	100	3
	software 6	36,731	43.65s	11	48	4
	software 7	138,305	38328.88s	34	147	47
	software 8	233,536	4285.13s	28	162	6
	software 9	47,268	2458.03s	25	273	1

[a] 참조 오류가 일어날 수 있는 배열의 수
[b] 오류가 일어날 수 있는 배열 참조식의 수

Table 1: 아이락의 분석 속도와 정확도

표2는 비용을 줄이는 여러 기술들을 사용했을 때 어떤 효과가 있는지를 보여준다. 이 표로부터 알 수 있듯이 데이터 스택 제거(stack obviation)가 가장 큰 효과를 발휘한다. 어떤 프로그램에서는 분석 지연시키기가 악영향을 미치기도 한다. 기다리는 지점 중 하나를 골라 분석을 계속해야 할 때 기다리는 지점 간의 계산 순서를 고려하지 않고 기다리던 지점 중에서 임의로 한 곳을 선택해서 분석을 계속 하기 때문이다.

	시간[a](초)	속도향상	시간[b](초)	속도향상
어떤 기술도 적용하지 않았을 때	18317.55	0%	16253.18	0%
선택적 합치기/비교하기	16055.58	12.35%	14286.72	12.1%
분석 지연시키기	19317.67	-5.45%	13153.43	19.98%
데이터 스택 제거	3717.06	79.71%	3247.79	81.02%
모든 기술을 적용했을 때	3461.57	81.11%	2320.58	85.73%

[a] 43개의 리눅스 커널 프로그램에 대한 분석 시간의 합
[b] a에서 분석 지연시키기가 악영향을 크게 미치는 하나의 프로그램을 제외한 시간

Table 2: 분석 속도 향상을 위한 기술들의 효과 비교

3 통계적 사후 분석

안전한 요약(sound approximation)을 계산해내는 분석기의 가장 큰 약점은 허위 경보(false alarm)라고 할 수 있다. 특히 안전한 분석을 제공하면서 분석 대상 프로그램에 대해 특별한 가정을 하지 않는 경우에는 실제 오류에 비해 허위 경보가 상대적으로 많을 수 있다. 허위 경보를 줄이기 위해 의미 공간(semantic domain)의 요약 수준(abstraction level)을 조절하는 것은 분석의 비용을 크게 증가시킬 수가 있다. 따라서 허위 경보에 대한 효과적인 대응방법이 되기 어려운 경우가 많다. 게다가 덜 요약된 의미 공간에서 더 정확한 분석 결과를 얻을 수 있다 하더라도 분석 대상 프로그램의 특성에 제한을 두지 않는다면, 분석 정확도를 크게 떨어뜨리는 프로그램이 항상 존재할 수 있다. 프로그램에 주석(program annotation)을 다는 방법은 분석 정확도 향상에 크게 기여할 수 있다[9]. 예를 들어 사용자가 프로그램의 특정 부분에 "이 지점에서 이 변수는 특정 범위의 값을 가진다"라는 식의 주석을 달 수 있을 것이다. 이런 주석이 있다면 분석기는 부정확한 분석 결과를 사용자 주석을 이용해서 보다 정확하게 수정할 수 있다. 경험적 지식(heuristics) 또한 분석기의 경보를 참 또는 거짓으로 구분하는데 활용될 수 있다[12]. 하지만 이론적 뒷받침없이 경험적 지식(heuristics)만을 이용한 분석 결과를 그대로 신뢰하기는 힘들다는 문제점이 있다. [12]의 예와 같이 경험적 지식(heuristics)을 이용해서 분석을 설계할 때 정확도 향상을 위해 분석의 안전성(soundness)를 포기하는 것을 고려하는 경우도 있다. 하지만 일부 오류만 빨리 찾아내는 것보다는 모든 오류를 찾아내는 것이 더 나으므로 항상 분석의 안전성을 포기하는 것은 바람직하지 않다. 아이락은 허위 경보 문제를 튼튼한 이론에 기반한 확률 기법의 사후 처리를 통해 해결함으로써 안전한 분석과 정확도 두 가지를 모두 해결하였다.

3.1 베이지안 분석

우리는 베이지안 분석(Bayesian analysis)[10]을 이용해 경보가 실제 배열 참조 오류(buffer overrun)를 가리킬 확률을 계산한다. $\oplus$를 발생한 경보가 참일 사건이라고 하고, $\ominus$를 허위 경보일 사건이라고 하자. S_i는 경보에서 관측된 증상(symptom)을 가리키고 $\vec{S}$는 이러한 증상들의 벡터(vector)이다. 이러한 증상들에는 어떤 것이 있고 어떻게 수집되는 지는 3.2 절에서 살펴보게 될 것이다. $P(E)$는 사건 E가 일어날 확률이고 $P(A \mid B)$는 사건 B가 일어났을 때 사건 A가 일어날 조건부 확률(conditional probability)이다. 증상 $\vec{S}$가 관측되었을 때 경보가

참일 확률, *'경보의 진실성(trueness of alarm)'*을 $P(\oplus \mid \vec{S})$라고 하자.

축적된 지식(knowledge base)으로 부터 새로운 사건의 확률을 계산할 때 베이즈 정리(Bayes' theorem)가 사용된다. 아이락에서 사용한 축적된 지식(knowledge base)은 미리 참/거짓으로 분류된 경보들로 구축했다. 미리 구분된 경보들은 교육 집합(learning set)이 되고 교육 집합으로부터 특정 증상(symptom)이 참인 경보와 허위 경보에 각각 나타나는 횟수를 셌다. 축적된 지식과 베이즈 정리를 이용하면 진실성 $P(\oplus \mid \vec{S})$는 다음의 식으로 계산할 수 있다:

$$P(\oplus \mid \vec{S}) = \frac{P(\vec{S} \mid \oplus)P(\oplus)}{P(\vec{S})} = \frac{P(\vec{S} \mid \oplus)P(\oplus)}{P(\vec{S} \mid \oplus)P(\oplus) + P(\vec{S} \mid \ominus)P(\ominus)}$$

또 $\vec{S}$의 각 증상이 나타날 사건이 독립 사건이라고 가정하면 다음을 얻을 수 있다:

$$P(\vec{S} \mid c) = \prod_{S_i \in \vec{S}} P(S_i \mid c) \text{ where } c \in \{\oplus, \ominus\}$$

$P(S_i \mid \oplus)$는 축적된 지식(knowledge base)을 바탕으로 계산된 $\hat{\psi}$을 이용해 추정(estimate)될 수 있으며 사전 분포(prior distribution)는 $[0, 1]$ 상에서 균등(uniform)한 것으로 가정한다. 발생한 경보 수 n에 대한 실제 오류 수의 비 $P(\oplus)$의 추정량(estimator)을 p라 하자. $P(S_i \mid \oplus)$와 $P(S_i \mid \ominus)$는 각각 θ_i와 η_i로 추정된다. 각 증상 S_i가 독립이라고 가정하면 축적된 지식(knowledge base)에 기반한 $P(\oplus \mid \vec{S})$의 사후 분포(posterior distribution)는 다음을 따른다:

$$\hat{\psi}_j = \frac{(\prod_{S_i \in \vec{S}} \theta_i) \cdot p}{(\prod_{S_i \in \vec{S}} \theta_i) \cdot p + (\prod_{S_i \in \vec{S}} \eta_i) \cdot (1 - p)} \tag{1}$$

여기서 p, θ_i, η_i는 다음의 베타 분포(beta distribution)를 가진다:

$$\begin{aligned} p &\sim Beta(N(\oplus) + 1, n - N(\oplus) + 1) \\ \theta_i &\sim Beta(N(\oplus,\ S_i) + 1, N(\oplus, \neg S_i) + 1) \\ \eta_i &\sim Beta(N(\ominus, S_i) + 1, N(\ominus, \neg S_i) + 1) \end{aligned}$$

$N(E)$는 축적된 지식(knowledge base)에 포함된 사건 E의 발생 횟수이다. 여기서 p, θ_i, η_i는 몬테카를로 기법(Monte Carlo method)으로 추정된다. 우리는 베타 분포로 부터 p_j, θ_{ij},η_{ij} 값을 임의로 N개씩 생성해서 이로 부터 N개의 ψ_j를 계산한다. 그러면 $\hat{\psi}$의 $100 \cdot (1-2\alpha)\%$에 해당하는 신뢰 집합(credible set)은 $(\psi_{j_{\alpha \cdot N}}, \psi_{j_{(1-\alpha) \cdot N}})$이 된다(단, $\psi_{j_1} < \psi_{j_2} < \cdots < \psi_{j_N}$). 이후에 설명하겠지만 참일 확률의 최대값이 우리의 관심사이므로, $\hat{\psi}$의 값으로 상계(upper bound) $\psi_{j_{(1-\alpha) \cdot N}}$를 취한다.

3.2 베이지안 분석을 위한 증상

우리는 분석의 정확도에 영향을 주는 요소들을 가능한한 모두 포괄하기 위해서 구문적 증상(syntactic symptom)과 의미적 증상(semantic symptom)을 정의하였다. 총 22가지의 증상을 정의하였고, 그중 12가지는 구문적 증상이고 10가지는 의미적 증상이었다. 증상들은 다시 세가지 부류로 나눌 수 있다: 1) 경보가 발생한 프로그램 식의 구문 문맥 (syntactic context)에 대한 것; 2) 정확도에 영향을 미치는 설계 상의 일반적 요소; 3) 배열 참조 식의 특성에 대한 것 등이다.

프로그램 식의 구문 문맥 경보가 발생한 프로그램 식이 어떤 구문 구조 속에 처해 있는 지를 설명하는 증상들이다. 중첩된 분기문이나 반복문 등을 많이 사용하는 경우 프로그램의 복잡도가 올라가게 되며 오류를 일으키기 쉽다. 이러한 문맥 속에 있는 프로그램 식에서 경보가 발생했다면 진짜 버그일 가능성이 높다고 할 수 있다.

정확도와 관련있는 일반적 요소 분석과정이나 설계와 밀접한 관련을 맺고 있는 증상들이다. 분석 과정에서 경보가 발생한 프로그램 식 이전에 있는 프로그램 포인트 들에서 최소 상계 연산(join)이나 축지법(widening) 등이 많이 일어났다면 분석의 정확도는 떨어진다고 볼 수 있다. 반면에 좁히기(narrowing)나 가지치기(context pruning) 등이 일어났다면 분석의 정확도가 향상된다고 볼 수 있다. 예를 들어 조건문 `x<10`은 변수 `x`의 값을 한정 짓는데 이용될 수 있다. 따라서 이런 류의 조건문은 이후 프로그램 포인트들에서의 분석 정확도를 향상시키는 요소라고 할 수 있다.

배열 참조 식의 특성 분석 결과의 정확도에 대한 증상이다. 축지법(widening)이 사용되는 경우 좁히기(narrowing)가 적절히 적용되지 못하면 분석 결과의 정확도는 매우 떨어질 수 있다. 예를 들어 경보가 발생한 배열 참조식의 인덱스(index) 변수가 $[-\infty, +\infty]$의 값을 가진다는 분석 결과를 얻은 경우와 $[0, 9]$의 값을 가진다는 결과를 얻은 경우 후자가 더 정확한 결과이므로 실제 오류일 가능성이 높다고 볼 수 있다. 이런 논리는 배열 인덱스(index) 뿐 아니라 배열의 크기에도 그대로 적용될 수 있다.

3.3 경보 걸러내기

추정된 확률을 이용해 참일 것 같지 않은 경보들을 체계적으로 걸러 낼 수 있다. 안전하지는 않지만(unsound) 보다 정확한 경보들만을 보고 싶은 사용자라면 확률을 이용한 경보 걸러내기를 사용할 수 있다. 경보를 걸러내기 위해서는 신뢰도(credibility) $100 \cdot (1 - 2\alpha)\%$로 추정된 확률 $\hat{\psi}$과 비교할 임계치가 필요하다. 아이락은 적절한 임계치를 얻기위해 사용자가 지정할 수 있는 두 가지 매개변수(parameter)를 제공한다. 실제 오류를 놓칠 수 있는 위험치 r_m과 허위 경보를 보고할 위험치 r_f를 사용자가 지정할 수 있다. 실제로는 둘 사이의 비(ratio)만 지정하면 된다.

	$\oplus$	$\ominus$
보고할 때의 위험치	0	r_f
보고하지 않을 때의 위험치	r_m	0

진실성(trueness)이 ψ인 경보가 주어졌을 때 경보를 보고할 때의 위험치에 대한 기대값은 $r_f \cdot (1 - \psi)$이 되고 보고하지 않았을 때의 위험치에 대한 기대값은 $r_m \cdot \psi$이 된다. 위험을 최소화 하기 위해서는 위험치가 작은 쪽을 택해야 할 것이다. 따라서 경보를 보고하기 위한 임계 확률은 다음과 같은 방법으로 선택할 수 있다:

$$r_m \cdot \psi \geq r_f \cdot (1 - \psi) \quad \Longleftrightarrow \quad \psi \geq \frac{r_f}{r_m + r_f}$$

만일 경보의 진실성(trueness)이 이 임계치보다 같거나 크면, 즉 진실성 $\hat{\psi}$의 최대값이 임계치보다 크면 경보는 신뢰도(credibility) $100 \cdot (1 - 2\alpha)\%$로 보고되어야 한다. 예를 들어 실제 오

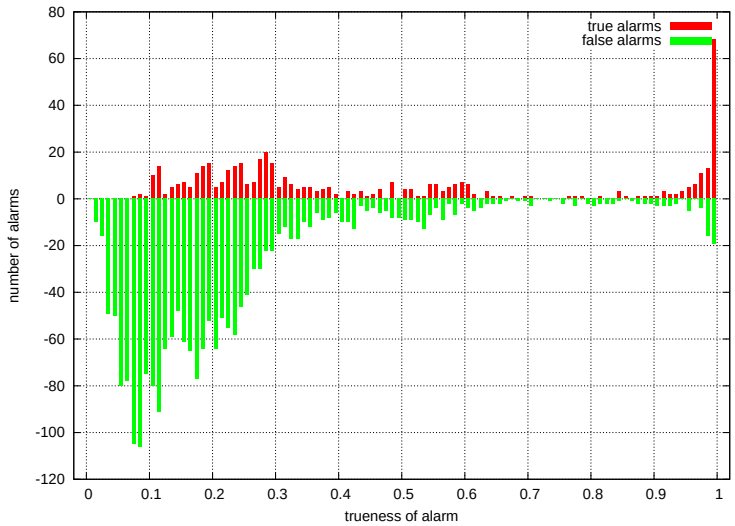


Figure 8: 확률별 경보수. 허위 경보의 수는 음수로 표시된다. 74.84%의 허위 경보가 참일 확률 0.25 미만의 확률을 가진다.

류를 놓칠 위험치가 허위 경보 위험치의 3배라고 생각하면, 사용자는 $r_m = 3, r_f = 1$과 같이 지정할 수 있다. 이 경우 경보의 보고여부를 판정할 임계치는 $1/4 = 0.25$가 되고, 참일 확률이 0.25 이상이 되는 경보만 사용자에게 보고된다.

우리는 리눅스 커널 소스 프로그램들과 알고리즘 교과서의 프로그램들을 이용하여 실험을 수행했다. 샘플 프로그램들의 분석 결과로 나온 경보들을 우선 교육 집합(learning set)과 테스트 집합(test set)으로 분류했다. 분류는 교육 집합과 테스트 집합의 비가 1:1이 되도록 했다. 교육 집합으로 분류된 경보들로부터 증상(symptom)들을 수집해서 경보가 참인 경우 허위인 경우에 나타난 횟수를 세었다. 이렇게 얻은 값을 바탕으로 테스트 집합에 속한 경보들의 ψ 값을 몬테카를로 기법으로 얻은 90% 신뢰 집합(credible set)에서 추정했다. 식 (1)을 이용해 2000개의 ψ_j를 2000개의 p, θ_i, η_i들로 부터 계산했다. 테스트 집합과 교육 집합은 겹치는 것이 없으므로 테스트 집합의 경보들은 모두 새로운 프로그램에서 발생한 것들로 볼 수 있다.

그림 8의 히스토그램(histogram)은 위의 실험을 15회 반복해서 얻은 것이다. 15회 반복실험에서 교육 집합과 테스트 집합의 분류는 매번 다르게 이루어졌다. 어두운 막대는 실제 오류에 해당하는 경보의 수를 흰 막대는 허위 경보의 수를 나타낸다. 74.83%(=1504/2010)의 허위 경보가 진실성(trueness) 0.25 미만이므로 사용자에게 보고되지 않을 수 있다. 즉 실제 오류를 놓칠 위험이 허위 경보의 위험의 3배라고 생각하는 사용자라면 허위 경보의 약 3/4을 제거할 수 있는 것이다.

분석의 안전성(soundness)을 위해서는 실제 오류를 놓치는 경우의 위험이 허위 경보의 위험보다 더 크게 봐야한다. 그러므로 $r_m \gg r_f$을 만족하도록 위험치를 지정하는 것이 보다 안전한 분석에 가깝다고 할 수 있다. 그림 8이 보여 주듯이 31.40%(=146/465)의 실제 오류에 대한 경보가 0.25 이하의 확률을 가지므로 허위 경보와 함께 보고되지 않을 수 있다. 비록 임계치를 0.07($r_m/r_f \simeq 13$)로 정하는 경우 실제 오류를 놓치지 않을 수는 있지만 이것은 우리의

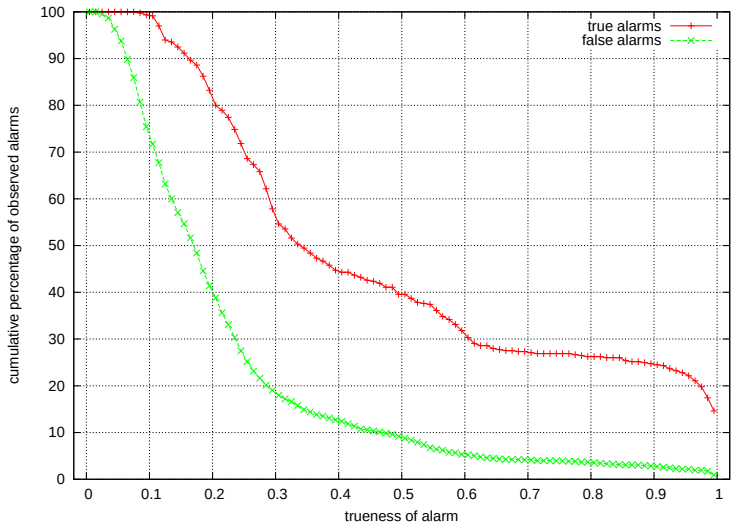


Figure 9: 참일 확률이 높은 경보부터 확인하면서 축적한 경보의 비율

실험에만 해당되는 수치로 분석의 안전성 보장과는 별 상관이 없다. 만일 경보를 모두 보고자 하는 경우라면 $r_f = 0$와 같이 설정하면 될 것이다.

3.4 경보 순위화

진실성(trueness)을 이용해 경보 보고를 진실성(trueness)이 높은 것부터 함으로써 사용자로 하여금 보다 효과적으로 경보 검증을 하도록 할 수 있다. 경보 순위화는 앞에서 이야기한 경보 거르기와 같이 혹은 별개로 사용할 수 있다. 실제 오류에 대한 경보의 진실성(trueness)은 0과 1 사이의 값을 가질 수 있지만 많은 허위 경보가 실제 오류에 대한 경보보다 낮은 진실성(trueness)을 가지고 있다. 따라서 진실성(trueness)으로 경보를 내림차순 정렬함으로써 사용자에게 실제 오류에 해당하는 경보를 보다 효과적으로 보고해 줄 수 있다. 그림 9는 그림 8의 경보들을 진실성(trueness)이 1인 경보부터 확인해 나갈때 확인한 경보들의 축적된 비율을 보여준다. 사용자는 실제 오류에 대한 경보의 50%(진실성 0.34)를 확인 하는 동안 15.17%(=305/2010)의 허위 경보만을 보게 된다.

증상(symptom)들을 바꿔가면서 시행해본 실험들을 통해 허위 경보와 실제 오류에 대한 경보들이 꽤 많은 증상들을 공유함을 확인했다. 이는 프로그램의 한 지점에서 발생한 오류가 구문 구조나 의미 구조와는 별 연관을 갖지 못하기 때문이 아닌가하는 의심을 들게 한다. 우리 실험에서 정의한 증상들 중 몇몇이 높은 변별력을 가지는 것으로 판단된다 하더라도 새 경보에서는 항상 기존 지식에 반하는 결과를 얻을 수 있다. 하지만 확률적 사후 처리는 보통의 프로그래밍 패턴에서 자주 나타나는 실수에 의한 오류나 분석기 자체의 취약성으로 인해 생기는 허위 경보들의 구분에는 잘 적용될 수 있는 것으로 보인다.

4 관련 연구

허위 경보(false alarm)를 줄이는 것은 언제나 정적 분석(static analysis)의 중요한 문제였다. 기존의 분석 도구들은 허위 경보에 대해 1) 안전하지 않은 분석을 수행하거나(SPLINT[16], ARCHER[15]); 2) 사용자 주석을 이용하거나(CSSV[9], SPLINT[16]); 3) 분석 대상 프로그램의 범위를 제한하거나(ASTRÉE[4, 8]); 4) 경험적 지식(heuristics)만을 이용한 경보 분류(Z-ranking[12])로 대응해 왔다.

아이락은 베이지안 확률 분석(Bayesian statistical analysis)을 이용해 허위 경보를 분류한다는 점에서 기존의 분석 도구들과 차별될 수 있다. 우리의 베이지안 확률 분석은 사용자 주석 등과 함께 독립적으로 사용될 수 있다. 아이락은 분석 자체로는 안전하면서, 사용자 주석에 의존하지 않고, ANSI C 프로그램 전체를 분석하며, 10,000줄의 코드를 한 번에 분석해 낸다.

CSSV[9]와 SPLINT[16]는 허위 경보를 줄이기 위해 사용자 주석에 의존한다. 이들 분석기의 경우 사용자 주석이 없거나 부정확한 경우 허위 경보의 비율은 급증한다. ARCHER(ARray CHeckER)[15]는 실제 오류를 모두 찾아 주지 않는 안전하지 않은 분석기이다. ASTRÉE[4, 8]는 항공 제어 분야의 C 프로그램에 버그가 없음을 검증하는 것을 목표로 하는 안전한 분석기이다. 이 분석기는 극소수의 허위 경보만을 발생시킨다. 하지만 ASTRÉE는 C 언어의 공용 구조체(union structure), 동적 메모리 할당(dynamic memory allocation), 제한 없는 재귀 함수 호출(recursive function call) 등은 지원하지 않는다.

우리의 베이지안 분석과 가장 관련이 깊은 기존 연구로는 Z-ranking[12]을 들 수 있다. Z-ranking은 경험적 지식(heuristics)을 바탕으로 경보들에 순위를 부여한다. 이를 위해 먼저 '성공(안전한 배열 참조)'과 '실패(배열 참조 오류 경보)'를 여러 그룹들로 뭉친다(partitioning). 각 그룹에 대해 경험적 지식(heuristics)을 이용해 각 경보가 참일 확률인 "z-score"를 계산한다. 경보들은 z-score값을 기준으로 내림차순으로 정렬된다. 이런 접근 방식은 두 가지 약점을 가진다. 경험적 지식(heuristics)은 각 그룹에 속한 '성공'과 '실패'의 상대적 수에 관한 것뿐이라는 것과 '성공'과 '실패'를 그룹들로 나누는 체계적인 방법이 없다는 것이다. 그러므로 실제를 반영하지 못하는 그룹별 구분은 유효한 z-score를 계산하기 어렵다. 우리가 사용한 베이지안 분석에는 Z-ranking의 임의적인 구분짓기 같은 것이 없다. 베이지안 분석에 이용된 경험적 지식(heuristics)이라 할 수 있는 증상(symptom)들은 유연한 확장성을 가지므로 Z-ranking에 비해 보다 경쟁력을 가진다고 할 수 있다. 베이지안 분석에 사용되는 증상들은 입력 프로그램 및 분석기 자체에 대해 정의할 수 있다. 마지막으로 베이지안 분석이 경험이 축적되면서 보다 정확한 확률을 계산해낼 수 있다는 사실은 아이락이 가지는 또 하나의 강점이 될 수 있다.

5 토의 및 결론

우리는 분석 대상 프로그램에 제한이 없는 안전한 분석의 설계와 베이지안 분석(Bayesian analysis)이 허위 경보(false alarm)에 대응하는 적절한 수단임을 보였다. 우리의 분석기 아이락은 모든 ANSI C 프로그램의 모든 배열 참조 오류(buffer overrun)를 자동으로 찾아주면서도 프로그램 분석 분야에 오랫동안 축적되어 온 여러 기술들을 활용하여 비용과 정확도의 균

형이 잡힌 분석을 제공한다. 정확도 향상에도 불구하고 안전성 보장으로 인한 불가피한 허위 경보들은 베이지안 통계 분석을 통한 사후 분석으로 대처하도록 하였다. 통계적 분석은 주어진 경보들이 참일 확률을 계산해 낸다. 이 확률들은 경보를 걸러내거나 정렬하는데 이용된다. 이렇게 함으로써 사용자가 보다 효율적으로 경보들을 검증할 수 있도록 해 준다. 리눅스 커널 프로그램과 알고리즘 교과서의 프로그램들을 이용한 실험에서 실제 오류를 놓칠 위험을 허위 경보의 위험보다 3배 크다고 지정한 경우 74.83%의 허위 경보를 걸러낼 수 있었고, 15.17%의 허위 경보만이 실제 오류 50%와 함께 섞여 있었다.

베이지안 분석의 유효성은 증상(symptom)들을 어떻게 정의하느냐에 달려 있다. 따라서 변별력 있는 증상들과 적절한 크기의 교육 집합이 베이지안 분석을 통한 접근에 핵심이 된다고 할 수 있다. 좋은 증상이라는 것은 분석 자체의 취약성이나 정확성과 밀접한 연관이 있는 것으로 보인다. 그러므로 분석에 대한 적절한 통찰이 좋은 증상을 정의하는데 필요하다고 본다. 그러나 구문적 증상은 특정한 프로그래밍 스타일을 반영하는 경향이 있고, 프로그래밍 스타일은 특정 조직 별로 다른 특성을 가질 수 있다. 그러므로 적절한 구문적 증상의 경우 특정 개발 환경 하에서 구축된 지식을 바탕으로 그 환경에서 개발된 프로그램들을 분석할 때 효과적인 증상이 될 수 있을 것이다. 베이지안 분석은 적절한 증상들을 정의한다면 어떤 분석과도 쉽게 어울려 사용할 수 있을 것으로 생각한다.

허위 경보를 다루는 또 다른 접근법은 정확도 향상에 기여할 수 있는 모든 기법들을 분석기에 장착하고 사용자로 하여금 분석할 프로그램에 따라 적절하게 필요한 기법들을 조합하여 쓸 수 있도록 하는 것이다. 적용 가능 기법들은 다양한 형태의 분석 대상 프로그램 별로 분석기를 특화할 수 있도록 확장성과 유연성을 가지고 있어야 한다. 이를 통해 분석기의 사용자는 허위 경보를 처리할 방법을 정할 수 있다. 한편 베이지안 분석을 이용한 접근법은 분석의 설계자로 하여금 분석기의 장단점을 바탕으로 증상을 정의할 수 있도록 한다. 이 두가지 방법을 적절히 함께 사용하지 못할 이유는 전혀 없다.

References

[1] bugtraq. `www.securityfocus.com`.

[2] CERT/CC advisories. `www.cert.org/advisories`.

[3] James O. Berger. *Statistical Decision Theory and Bayesian Analysis, 2nd Edition.* Springer, 1985.

[4] Bruno Blanchet, Patric Cousot, Radhia Cousot, Jerome Feret, Laurent Mauborgne, Antonie Mine, David Monnizux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–207, June 2003.

[5] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

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

[7] Patrick Cousot and Radhia Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP '92: Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pages 269–295. Springer-Verlag, 1992.

[8] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astrée analyzer. In M. Sagiv, editor, *European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer-Verlag, 2005.

[9] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cssv: towards a realistic tool for statically detecting all buffer overflows in c. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 155–167. ACM Press, 2003.

[10] Andrew Gelman, John B. Carlin, Hal S. Stern, and Donald B. Rubin. *Bayesian Data Analysis*. Text in Statistical Science. Chapman & Hall/CRC, second edition edition, 2004.

[11] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[12] Ted Kremenek and Dawson Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In Radhia Cousot, editor, *SAS '03: Proceedings of the 10th Annual International Static Analysis Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 295–315. Springer, 2003.

[13] Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In M. Sagiv, editor, *European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 5–20. Springer-Verlag, 2005.

[14] N. Metropolis and S. Ulam. The Monte Carlo method. *Journal of the American Statistical Association*, 44(247):335–341, September 1949.

[15] Yichen Xie, Andy Chou, and Dawson Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 327–336. ACM Press, 2003.

[16] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 97–106. ACM Press, 2004.