자동 오류 검출을 위한 프로그램 분석기 - 아이락

정영범, 김재황, 신재호, 이광근
{dreameye,jaehwang,netj,kwang}@ropas.snu.ac.kr
서울 대학교 프로그래밍 연구실

[마이크로 소프트웨어], pp.178-186, 2005년 6월

1 프로그램 오류를 해결하는 기술들

굳이 유비쿼터스(ubiquitous)를 말하지 않아도 우리는 숨쉬는 일만큼 자연스럽게 언제 어디서나 컴퓨터를 만날 수 있는 세상에서 살고 있다. 과연 우리는 이전 보다 더 편하고 안전한 세상을 살고 있는 것일까? 자신 있게 "그렇다"라고 말하기에는 맘속에 이는 약간의 불안감을 떨쳐내기 힘들다. 그 불안감의 이유는 여러 가지가 있겠지만, 그 중에서도 가장 먼저 떠오르는 것을 한 가지 꼽으라면 프로그램의 오류, 즉 버그(bug)를 꼽을 수 있겠다.

현대인들이 생활의 많은 부분을 의존하고 있는 컴퓨터 위에서 돌아가고 있는 수많은 프로그램들이 정말 버그 없이 안전하게 돌아갈 것이라는 것을 아무도 보장하지 못하는 것이 현실이다. 때문에 위험은 우리가 볼 수 없는 프로그램의 어두운 부분에서 항상 도사리고 있다가 예상치 못하는 때에 튀어 나와 사고를 일으킨다. 지구 상 곳곳에 존재하는 컴퓨터 위에서 돌아가는 프로그램들안에 버그들이 우글거리고 있다고 상상해보라. 얼마나 끔직한 일이 될 것인가?

다행히도 컴퓨터의 발달과 함께 오랜 시간에 걸쳐 프로그램의 버그를 줄일 수 있는 많은 기술들이 개발되어왔다. 그 기술들 중에 1970년대 제일 처음 모습을 드러낸 것이 바로 프로그램이 제대로 생겼는지를 검증해 내는 구문검증기술(parsing)이다. 이 기술은 이미 우리가 너무도 익숙하게 사용하고 있어서 지금은 거의 관심을 두지 않는 기술이다. 이 기술 덕분에 우리는 더 이상 우리가 짠 프로그램의 생김새를 일일이 눈으로 확인하지 않아도 되게 되었다.

그런데 생긴 것은 멀쩡한데 실제로 제대로 돌지 않는 프로그램을 걸러낼 필요성이 자연스레 생겨났다. 예를 들어 x = "버그" + 10 같은 프로그램은 생긴 것은 멀쩡하지만 하는 일은 전혀 멀쩡하지 않은 프로그램이다. 이런 프로그램들을 안전하게 걸러내기 위해서 1990년대에 활성화된 기술이 바로 타입 검증(type checking)이다. 타입에 맞지 않는 프로그램들은 실행 중에 계산을 할 수 없게 되어 예측할 수 없는 방향으로 프로그램이 실행되거나 혹은 원인도 모르게 비정상 종료한다. 따라서 프로그램의 모든 값들은 항상 자신들이 정의된 타입에 맞는 값들을 가져야 한다. 어떤 프로그램이 항상 그러한 성질을 만족하면서 실행 될 것인지를 자동으로 검증해 주는 것이 바로 타입 검증 기술이다. 사실 이 기술이 모든 프로그래밍언어 전반에 잘 스며들어 있지는 않다. 아직까지는 가장 널리 쓰이는 C 언어에서는 강제형변환(type casting)을 통해 타입을 억지로 맞추어, 실제로는 문제가 되는 코드도 안전한 것처럼 컴파일러의 타입체크를 통과 시킬 수도 있다. 앞으로 더욱 더 많은 프로그래밍언어로 퍼져나가야 하는 기술이다.

한편, 타입이 맞는 프로그램이라고 해서 모두 제대로 돈다고 할 수는 없다. 타입보다 더욱 정교한 성질에 대한 보장이 필요하다. 물을 만들기 위해서는 수소와 산소가 와야지 수소랑 질소가

와서는 안 된다는 사실을 검증하는 것은 타입검증과 비슷하다. 하지만, 물 100g 이상을 만들기 위해서는 적어도 수소 몇 g과 산소 몇 g이 필요하다는 사실까지 검증은 보다 정교한 기술을 필요로 한다. 프로그램의 버그 잡는 기술 중에 지난 이십년간 발전되어 온 자동증명기술(theorem proving)과 정적분석기술(static analysis)이 바로 그러한 정교한 기술들의 예이다. 이러한 기술들은 우리가 분석하고자 하는 프로그램의 성질을 정교한 논리식으로 정의하고 그 식을 만족하는지를 온갖 방법을 동원해 찾아낸다. 대게 이러한 정교한 분석들은 많은 시간과 자원을 필요로 하기 때문에 아직은 컴파일러에 장착될 정도는 되지 못한다. 현재로선 이 기술이 무르익어서 모든 사람들이 아무렇지 않게 사용하게 될 날은 아직 멀어 보이지만 지금 이 순간에도 전 세계의 수많은 그룹들이 이 기술을 더 견고히 발전시키기 위해 천천히 나아가고 있다.

2 정적 프로그램 분석(static program analysis)

정적 프로그램 분석(static program analysis)[5]라는 것은 프로그램의 성질을 실행 전에 자동으로 안전하게 빠짐없이 예측하는 기술이다. 사실 프로그램의 성질(프로그램 오류)을 실행 전에 정확히 알아내기란 불가능하다. 그 유명한 halting problem이 그 불가능성을 시사한다. 하지만, 프로그램 분석(program analysis) 분야에서는 오랜 기간동안 불가능에 도전해 왔고, 어느 정도는 프로그램의 성질을 알아내는 것이 가능하게 되었다.

예를 들어 다음 날에 비가 오는 것을 예측하는 기계가 있다고 해보자. 그런데 다음 날 비가 올지 안 올지 정확하게 예측하는 것은 불가능하다고 하자. 그러면 우리가 만들 수 있는 "안전한" 기계는 바로 다음과 같이 동작하는 기계이다. "기계가 다음날 비가 오지 않는다고 말하면 실제로도 항상 다음 날 비가 오지 않는다." 단순히 이러한 기계를 만드는 것은 아주 쉽다. 항상 내일 비가 온다고 말하는 기계를 만들면 된다. 그런데 이런 기계는 사실 의미가 없다(아무런 정보도 주지 않으므로). 우리가 할 수 있는 최선은 실제로 다음 날 비가 오지 않는다는 것이 확실한 날들에 대해서는 오지 않는다는 이야기를 많이 해주는 그런 기계를 만드는 것이다. 우리가 하는 프로그램 분석(program analysis) 기술도 마찬가지이다.

안전하고 정확한 분석기란 버그가 있을 것 같은 곳에서는 항상 있을 지도 모른다는 말을 해주면서 실제로 없는 곳은 많이 피해가는 그런 분석기이다. 버그가 있을 만한 곳에서는 모두 말을 해주기 때문에 "빠짐없는" 분석기이다. 어떤 방법이 이러한 기술을 가능하게 하는 지 살펴보자. 알고 보면 정말 간단하다. 프로그램이 실행 중에 일어날 수 있는 모든 일을 예측할 수 있으면 된다. 모든 프로그램 포인트에서 일어날 수 있는 모든 프로그램의 상태들을 계산해 내면 되는 일이다. 다음의 간단한 코드를 보자.

```
int main( )
{
  int x, idx;
  x = 0;
  while (x<100) {
    idx = rand()%10;
    x = x + 1;
  }
  return 0;
}
```

우리가 알고 싶은 프로그램의 성질이 만약 이 프로그램이 끝나기 전에 메모리에 있는 변수들이 가지고 있는 값들이라고 생각을 해보자. 그럼 실제로 실행을 해보면 어떻게 될까? 변수 x에 저장된 값은 루프(loop)를 돌고 난 후의 값이니 100이 되어 있을 테고, 변수 idx에 저장된 값은 0부터 9사이의 임의의 값을 하나 갖게 될 것이다. 그렇다면 정적 프로그램 분석(static program analysis)의 결과는 무엇이 되어야 안전하겠는가? 변수 i의 값은 100이고 변수 idx의 값은 0에서부터 9사이의 모든 값이라고 하면 안전한 분석이다. 실제 실행 중에 일어 날 수 있는 모든 것을 포섭하고 있기 때문이다. 물론 더 많은 것을 이야기해도 안전하다. 예를 들면 변수 i의 값은 100이상이고 idx는 모든 값이 될 수 있음이라고 이야기하는 것이 되겠다. 하지만, 이 것은 첫 번째 분석 결과보다 더 많은 집합을 이야기 하고 있으므로 부정확하다. 우리의 분석결과로 정확히 알고 싶은 것이 2라는 값이었다면 정수, 짝수, 0보다 큰 짝수, 0하고 10사이의 짝수라고 이야기하는 것들은 안전하지만 부정확한 분석결과이다.

정적 프로그램 분석(static program analysis)과 테스트나 실행 중에 검사(runtime check)하는 것과의 차이는 무엇일까? 가장 중요한 차이를 말하면 정적 프로그램 분석은 실행 중에 일어날 수 있는 모든 것을 포섭하지만 테스트나 실행 중에 검사는 그렇지 못하다는 것이다. 위의 코드만 보더라도 실제 실행을 해보면 10번의 테스트를 하더라도 변수 idx가 가지는 값들을 전부 찾아낸다는 보장을 절대 할 수 없다. 위와 같이 간단한 코드에서조차도 힘들진대 실제 현장에서 개발하고 있는 커다란 프로그램에 대해서는 어떻겠는가? 또, 한 번 실행을 하는데 오랜 시간이 걸리거나 프로그램의 입력으로 될 수 있는 값들의 개수가 무한하거나 혹은 끝나지 않는 프로그램들을 테스트를 통해서 검증해내기가 정말 어려운 일이다. 그런 면에서 테스트나 실행 중 검사는 한계를 지니고 일반적으로 프로그램에 버그가 없다는 것을 보장할 수 없다.

실제로 프로그램 분석 기술을 사용하여 유용한 일을 해낸 외국의 사례들을 살펴보자. 윈도우즈 98을 사용하던 대부분의 많은 사람들은 이유를 모른 채 발생하는 블루스크린에 짜증이 난 경험이 있을 것이다. 윈도우즈 XP에서는 블루스크린을 거의 볼 수 없게 되었는데 그 이면에는 마이크로소프트사 내부에서 버그 잡는 기술에 대한 투자가 있었다. 그 한 예가 바로 SLAM[4] 프로젝트이다. SLAM은 Microsoft 연구소에서 프로그램이 원하는 속성을 만족하는지 검증하고 버그를 발견해 주어 신뢰성 높은 소프트웨어를 제작토록 도와주기 위한 프로젝트이다. Microsoft는 SLAM 프로젝트의 결과물을 이용하여 Microsoft Windows XP가 사용하는 디바이스 드라이버(device driver)의 버그들을 찾아내고 있다. 윈도우즈의 많은 버그들 중에는 드라이버에서 발생하는 버그들이 많은데, 왜냐하면 이런 드라이버들은 새로운 기기가 추가될 때마다 새로이 작성되고 개발자들이 보통 운영체제를 개발한 쪽과 다르기 때문이다. SLAM의 결과물로 현재 100 여 개의 드라이버에 대해 20 여 개의 속성을 검증하여 20 개 이상의 버그를 찾아냈다고 한다. Bandera[3]는 Java로 작성된 프로그램이 올바르게 작동하는지를 검증해주는 프로그래밍 도구이다. 특히 여러 쓰레드(thread)가 동시에 (concurrent) 수행되는 경우에 발생 가능한 오류를 찾아준다. 예를 들어, 모든 쓰레드가 서로를 기다리기만 하는 교착 상태(dead lock)에 빠지지 않는지, 또는 무한정 어떤 조건이 만족되기를 기다리기만 하는 쓰레드가 없는지 검사해 준다. Bandera는 Kansas 주립 대학에서 1998년부터 개발해왔고 실제 현장에 적용되고 있다. NASA와 Honeywell technology center에서 Bandera를 사용하여 개발하는 소프트웨어를 검증하고 있다. 특히 Honeywell technology center는 Bandera를 이용하여 항공 운항을 위한 실시간 운영체제인 DEOS(Digital Engine Operating System)에서 오류를 찾아냈고 수정할 수 있었다. 또 다른 예로 AirBus의 콘트롤러 소프트웨어 모듈 검증이 프로그램 분석 기술을 통해서 성공적으로 이루어졌다[2]. 기존에 의존하던 다른 검증 기술(verification by theorem proving)보다 그 비용과 효과가

뛰어났기 때문에, AirBus에서는 비행기 내장 소프트웨어 개발에 프로그램 분석기술을 통한 검증 단계가 반드시 들어가도록 표준 개발과정을 개편하고 있다.

3 요약해석(abstract interpretation) 개요

초등학교 교실의 산수 시간. 선생님이 정수식 계산 문제 하나를 학생들에게 내준다.

문제 : 128 * 22 + (1920 * (-10)) + 4 = ?

0.1 초 후에 철호가 정수라는 답을 제시하는 것을 시작으로 시간이 흐름에 따라 영호, 진호, 명호, 상호가 점점 더 정확한 답을 얘기한다.

철호 : 정수
영호 : 짝수
진호 : 음수
명호 : -100000보다 크고 10000보다 작은 정 수
상호 : -16380

철호, 영호, 진호, 명호, 상호 등의 답은 모두 옳다고 할 수 있다. 다만 정확도에는 차이가 난다. 모두 옳다는 얘기는 각각이 제시한 답 속에 정확한 답 -16380이 포함되기 때문이다. 정확도의 차이는 각 답들이 제시하는 집합의 크기에서 기인한다. 마지막 상호의 답이 가장정확한데 상호의 답은 하나의 원소만 가지고 있기 때문이다. 짝수라는 답을 얘기한 영호의 머리 속을 잠시 들여다보자. 영호는 모든 정수는 홀수와 짝수의 집합의 한 원소에 대응 시킬 수 있다는 사실을 알고 있었을 것이다. 그리고 짝수와 짝수의 합은 짝수, 짝수와 짝수의 곱은 짝수, 홀수와 짝수의 합은 홀수가 된다는 식의 홀수, 짝수 계산법을 알고 있었을 것이다. 영호의 머리 속에서 주어진 문제는 "짝수 * 짝수 + 짝수 * 짝수 + 짝수"라는 문제로 바뀐 후, 알고 있는 홀수, 짝수 계산법이 적용되어 최종적으로 짝수라는 답에 이르게 되었을 것이다.

요약 해석이라는 것은 영호가 했던 일을 컴퓨터 프로그램에 적용하는 것이라고 볼 수 있다. 주어진 프로그램이 어떻게 실행될 지를 정의해 주는 것을 프로그램의 의미(semantics)라 한다. 프로그램의 실행을 정의하는 방법은 다양하지만 어떤 방법이든지 프로그램의 실행을 정의하는 데 필요한 값들이 돌아다니는 도메인(domain)을 필요로 한다. 예를 들면, 계산식을 위해서는 정수 도메인(domain)이 필요하고, 포인터를 다루기 위해서는 메모리 주소를 원소로 가지는 도메인(domain)이 필요할 것이다. 또 대입문(assignment statement)을 위해서는 주소에 값을 대응시키는 함수들의 집합인 메모리라는 도메인(domain)이 있어야 할 것이다.

요약해석(abstract interpretation)은 주어진 프로그램을 프로그램의 실제 실행을 정의하는 도메인(domain) 대신에 요약된 도메인(domain), 예를 들어 정수의 요약 도메인으로서의 홀수, 짝수 집합에서 실행하면서 실제 프로그램이 실행 중에 가질 수 있는 값들을 모두 포섭하는 결과 값을 계산해내는 분석이라고 할 수 있다. 만일 요약 해석이 요약 도메인(domain) 상에서의 프로그램 실행에만 관한 내용이라면 요약 해석은 하나의 아이디어에 지나지 않을 것이다. 요약 해석이 훌륭한 이유는 요약된 도메인(domain)에서 얻은 프로그램 실행의 결과가 실제 프로그램 실행의 결과를 온전하게 포섭하는 것이 보장되는 올바른 분석을 디자인 할 수 있는 프레임웍을 제공해 주기 때문이다. 영호가 주어진 정수식의 답으로 짝수를 제시했는데, 이것이 옳다는 것(직관적으로는 옳지만)은 어떻게 증명할 수 있을까? 모든 정수식에 대해서 영호식의 계산법으로 얻은 답이 실제 계산식의 계산 결과를 포섭함을 증명하는 것은 답이 뻔히 보이는 간단한 문제는 아니다.

4 아이락 설계

아이락[1]은 이러한 요약 해석의 틀 속에서 설계되고 구현되었다. 요약 해석의 틀 속에서 아이락이 프로그램의 실제 실행을 어떻게 요약하는 지 예를 통해서 살펴보자. 아이락은 주어진 프로그램이 잘못된 배열 참조를 하는 경우가 있는 지 없는 지를 분석하는 것을 목표로 하고 있다. 그러므로 아이락이 사용하는 요약 도메인(domain)은 영호의 것처럼 홀수, 짝수만으로 이루어져있어서는 제대로 된 분석 결과를 내놓지 못할 것이다. 아이락의 분석 목표에 맞는 수준의 요약 도메인(domain)이 필요한 것이다.

아이락은 프로그램에 있는 수식 값들의 요약 도메인(domain)으로 인터벌(interval)이라는 도메인(domain)을 이용한다. $[a, b]$로 표시되는 인터벌 도메인(domain)의 원소는 a보다는 크거나 같고, b보다는 작거나 같은 수들의 집합을 의미한다. 또 특별히 기호를 이용해서 크기가 무한한 도메인(domain)을 나타낼 수도 있다. 즉 $[-\infty, \infty]$은 모든 수 전체를 의미하는 인터벌이 된다. 인터벌 도메인(domain)의 원소 간에는 순서(order)도 정의할 수 정의할 수 있다. 인터벌 I, J 간의 순서 $\sqsubseteq$는 인터벌 I가 인터벌 J에 포함되는 경우 $I \sqsubseteq J$로 나타낸다. 그리고 모든 인터벌보다 작은 인터벌을 특별히 기호 $\perp$로 나타낸다.

배열은 배열의 요약된 시작 주소, 배열의 크기 인터벌, 그리고 현재 포인터가 가리키고 있는 배열의 오프셋(offset) 인터벌을 가지는 값으로 요약된다. 메모리는 메모리가 할당된 프로그램 상의 위치로 요약된다. 이를 위해 아이락은 프로그램의 모든 식, 명령문, 선언 등에 고유한 레이블을 할당한다. 다음의 프로그램을 생각해보자.

```
int a[10], *p;
p = a + 3;
```

아이락의 요약 도메인(domain)에서 이 프로그램을 실행하고 나면 a는 $\langle l, [10, 10], 0\rangle$을, p는 $\langle l, [10, 10], [3, 3]\rangle$을 가지게 된다. 여기서 l는 배열 선언 a[10]에 할당된 레이블이다. 아이락은 프로그램의 모든 배열의 크기와 그 배열 접근에 이용되는 인덱스의 값을 인터벌로 요약한 후, 인덱스 인터벌이 크기 인터벌 외의 값을 가질 가능성이 있을 때 버퍼오버플로우가 날 수 있다는 알람을 발생 시킨다.

앞에서 예로 든 프로그램의 일부를 담고 있는 다음의 프로그램을 통해 아이락이 프로그램을 인터벌로 요약하는 과정을 알아보자.

```
x = 0; ①
while(x<10) {
  ② x = x+1; ③
} ④
```

이 프로그램의 실행으로 프로그램의 각 지점(프로그램 포인트, program point)에서 변수 x가 가질 수 있는 값(인터벌)은 무엇이 될까? 상식적으로 접근해보자. 우선 프로그램 포인트 ①에서 x는 x = 0;의 실행 결과로 0을 값으로 가질 것이다. 프로그램 포인트 ②에서는 프로그램 포인트 ①과 같은 값을 가질 것이고, 프로그램 포인트 ③에서는 x = x +1;의 실행 결과로 프로그램 포인트 ②보다 1 큰 값을 가질 것이다. 프로그램 실행 흐름은 여기까지 이른 후에 다시 프로그램 포인트 ①로 돌아가므로 프로그램 포인트 ①에서 x는 이제 0 또는 1의 값(인터벌로 표현하면 $[0, 1]$)을 가진다고 할 수 있다. 이런 식으로 생각을 해보면 x는 프로그램 포인트 ①에서는 $[0, 10]$, 프로그램 포인트 ②에서는 $[0, 9]$, 프로그램 포인트 ③에서는 $[1, 10]$ 마지막으로 프로그램 포인트 ④에서는 $[10, 10]$의 값을 가진다고 할 수 있을 것이다. 이 과정을 구현한 프로그램 분석기는 어떻

게 만들 수 있을까? 이렇게 접근해 보자. 우선 우리가 알고 싶은 것, 즉 각 프로그램 포인트에서 변수 x가 가질 수 있는 값을 미지수로 가지는 방정식을 세우자. 그리고 인터벌 도메인(domain) 상에서 이 방정식을 풀자. 그러면 그 방정식의 해가 바로 우리가 원하는 분석 결과가 될 것이다. 각 프로그램 포인트 에서 x가 가지는 값을 X_i라는 미지수로 표현하자. 프로그램의 실행 의미를 생각하면 다음과 같은 방정식을 만들어 갈 수 있을 것이다. 우선 자연어로 방정식을 표현해 보자.

$$\begin{array}{lcl} X_1 & = & [0,0] \text{ 또는 } X_3 \\ X_2 & = & X_1 \text{ 이면서 } 10 \text{ 미만} \quad = \quad X_1\text{에 포함되고 } [-\infty, 9]\text{에도 포함됨} \\ X_3 & = & X_2\text{에 } [1,1]\text{을 더함} \\ X_4 & = & X_1\text{이면서 } 10 \text{ 이상} \quad = \quad X_1\text{에 포함되고 } [10, \infty]\text{에도 포함됨} \end{array}$$

우리가 원하는 분석은 프로그램의 실제 실행을 포섭하는 것이어야 한다는 점과 분석 결과가 실제를 포섭하면서도 결과가 나타내는 집합의 크기가 작을수록 분석의 정확도가 높다는 점을 상기하자. "$X_1 = [0,0]$ 또는 X_3"에서 좌변을 $[-\infty, \infty]$로 바꿔도 틀리지는 않다. 하지만 정확도는 가장 낮다고 할 수 있다. 왜냐하면 $[-\infty, \infty]$은 모든 수의 집합을 의미하기 때문이다. 그러면 "$[0,0]$ 또는 X_3"을 포섭하면서 가장 작은 값은 어떻게 나타내면 좋을까? 수학시간에 배운 합집합이 바로 우리가 원하는 답이다. 두 집합의 합집합은 둘을 포함하는 최소 집합이므로. 집합세계의 합집합을 인터벌로 확장해서 $\sqcup$이라는 기호로 쓰자. 예를 들어 의 결과는 이다. "X_1에 포함되고$[-\infty, 9]$에도 포함됨"은 집합세계의 교집합을 이용해서 나타낼 수 있을 것이다. 기호로는 $\sqcap$을 사용하자. 예를 들어 $[1,2] \sqcap [2,3]$의 결과는 $[2,2]$가 될 것이다. 인터벌 간의 덧셈은 $\oplus$로 쓰고, $[a,b] \oplus [b,c] = [a+c, b+d]$로 정의하자. 이 정의는 각 인터벌에 포함되는 수 간의 덧셈 결과를 포함하는 최소 인터벌을 의미한다. 이제 자연어로 표현했던 방정식은 다음과 같은 모양이 된다.

$$\begin{array}{lcl} X_1 & = & [0,0] \sqcup X_3 \\ X_2 & = & X_1 \sqcap [-\infty, 9] \\ X_3 & = & X_2 \oplus [1,1] \\ X_4 & = & X_1 \sqcap [10, \infty] \end{array}$$

자 이제 어떻게 풀어야 할지는 모르겠지만, 모양은 그럴 듯 한 방정식을 얻었다. 이제 우리의 목표는 이 방정식의 해 중에서 인터벌 도메인(domain)의 순서 기준으로 가급적 작은 해를 구하는 것이다. 우리의 방정식을 조금 다르게 표현해 보자.

$$\begin{array}{rcl} (X_1, X_2, X_3, X_4) & = & F(X_1, X_2, X_3, X_4) \\ F(X_1, X_2, X_3, X_4) & = & ([0,0] \sqcup X_3, X_1 \sqcap [-\infty, 9], X_2 \oplus [1,1], X_1 \sqcap [10, \infty]) \end{array}$$

방정식을 위와 같이 바꿔 쓰면 우리가 원하는 해는 함수 F의 고정점(fixpoint)이라고 할 수 있다. 함수의 고정점이란 함수의 입력과 출력이 같게 되는 값을 의미한다. 즉, 함수 f에 대해 $f(x) = x$를 만족하는 x가 함수 f의 고정점이다. 만일 이런 x가 둘 이상 있고 그들 간에 순서가 있다면 그 중에서 가장 작은 값을 최소고정점(least fixpoint)라고 한다. 특정 조건을 만족하는 경우 함수의 고정점을 구하는 기계적인 방법이 존재한다. 그 방법은 함수가 정의된 도메인(domain)의 최소 원소에 함수 적용 값이 변하지 않을 때까지 함수를 계속 적용하는 것이다. 함수 F의 k번 적용을 의미하는 F^k를 다음과 같이 정의하자.

$$
\begin{aligned}
F^0(\bot,\bot,\bot,\bot) &= (\bot,\bot,\bot,\bot) \\
F^{k+1}(\bot,\bot,\bot,\bot) &= F(F^k(\bot,\bot,\bot,\bot)) \quad (k>0)
\end{aligned}
$$

우리의 경우 $F^{k+1}(\bot,\bot,\bot,\bot) = F^k(\bot,\bot,\bot,\bot)$ 이 될 때까지 $k+1$번 함수 F를 적용하면 된다. 다음의 표는 함수 F를 적용해 가는 과정을 보여주고 있다.

	0	1	2	3	...	12	13	14
X_1	$\bot$	[0,0]	[0,0]	[0,1]	...	[0,10]	[0,10]	[0,10]
X_2	$\bot$	$\bot$	[0,0]	[0,1]	...	[0,9]	[0,9]	[0,9]
X_3	$\bot$	$\bot$	[1,1]	[1,2]	...	[1.10]	[1,10]	[1,10]
X_4	$\bot$	$\bot$	$\bot$	$\bot$	...	$\bot$	[10,10]	[10,10]

이 표는 k가 14인 경우 13인 경우와 다른 값을 가지는 X_i의 값이 없음을 보여준다. 즉 함수 F를 13번 적용해서 고정점에 도달했으며 이를 14번째 적용에서 확인 한 것이다. 최종 결과가 앞서의 상식적인 접근으로 얻은 결과에 부합함을 확인하자.

이제 루프의 조건이 외부 입력으로 결정되도록 분석 대상 프로그램을 조금 바꿔 보자.

```
n = readint()
x = 0; ①
while(x<n) {
  ② x = x+1; ③
} ④
```

변수 n의 값은 외부 입력으로 결정되므로 어떤 값이든지 될 수 있다. 인터벌로는 $[-\infty,\infty]$에 해당한다. 이 프로그램에 대한 방정식은 다음과 같다.

$$
\begin{aligned}
X_1 &= [0,0] \sqcup X_3 \\
X_2 &= X_1 \sqcap [-\infty,\infty] = X_1 \\
X_3 &= X_2 \oplus [1,1] \\
X_4 &= X_1 \sqcap [-\infty,\infty] = X_1
\end{aligned}
$$

이 방정식의 답을 구하는 계산은 끝이 나지 않는다. 함수를 $(k>1)$번 반복 적용한 후의 X_0은 $[0,k-2]$가 된다. 보통의 방법으로는 고정점에 도달하지 못하는 것이다. 어떻게 하면 될까? 우리의 목표가 실제 일어날 수 있는 상황과 정확히 일치하는 분석 결과를 얻는 것이 아니라 실제 상황을 포섭하는 분석결과를 얻는 것이라는 점을 생각하면 해결책을 구할 수 있다. 해결책은 방적식의 답을 포섭하면서 인터벌이 커질 수 있는 횟수를 제한하는 것이다. 이를 넓히기(widening)이라고 한다. X_1에 대해 $[0,0]\sqcup[1,1]$의 값으로 $[0,1]$대신에 $[0,\infty]$을 취한다면 다음번 함수 적용에서도 $[0,\infty]$이 되므로 함수 적용을 멈출 수 있게 된다. 이런 넓히기를 적절히 이용하면 어떤 프로그램이 주어지더라도 항상 유한한 시간 내에 분석을 완료할 수 있게 된다. 하지만 넓히기를 불필요하게 많이 사용하면 분석의 정확도는 떨어지게 된다. 이를 방지하기 위해서는 넓히기를 적용하는 경우를 최소화 하는 방법과 넓히기로 인해 떨어진 정확도를 보정할 방법이 필요하다. 넓히기는 while등의 반복문, 재귀함수 등으로 생기는 루프의 시작점에서만 적용하면 된다. 사실 루프의 반복 횟수가 유한하다면 굳이 넓히기를 사용하지 않아도 유한한 시간 내에 분석을 끝낼 수가 있다. 그러나 루프의 반복 횟수가 유한 할 지의 여부를 결정할 일반적인 방법은 없기 때문

에, 모든 루프에 대해 넓히기를 적용해야만 항상 유한한 시간 내에 분석이 끝남을 보장할 수 있다. 그리고 여기서는 소개하지 않겠지만, 넓히기 후에 좁히기(narrowing)를 적용하면 넓히기로 떨어진 정확도를 상당 부분 보정할 수 있다.

아이락은 프로그램의 요약 도메인(domain) 설계, 프로그램으로부터 방정식을 도출하는 방법과 방정식의 해를 구하는 방법 등을 요약 해석이 제시하는 프레임웍 내에서 설계하고 구현하였다. 따라서 아이락은 "프로그램의 분석 결과가 프로그램의 실제 실행에서 일어날 모든 상황을 포섭함"을 보장받게 된다.

5 분석 예

이제, 아이락이 실제 소프트웨어에서 찾아내는 오류들을 살펴보자.

5.1 Linux kernel 2.6.4 장치 드라이버

Linux kernel 2.6.4의 USB 모뎀 드라이버와 관련된 /drivers/usb/class/cdc-acm.c 파일의 일부분이 다음과 같다.

```
static struct acm *acm_table[ACM_TTY_MINORS];
...

static int acm_probe (struct usb_interface *intf,
                      const struct usb_device_id *id)
{
...
    for (minor = 0; minor < ACM_TTY_MINORS && acm_table[minor]; minor++);
    if (acm_table[minor]) {
            err("no more free acm devices");
            return -ENODEV;
    }
...
    acm_table[minor] = acm;
...
}
```

acm_table이라는 배열이 ACM_TTY_MINORS의 크기로 선언되어있다. acm_probe라는 함수의 한 부분에서 for를 돌면서 acm_table에서 0인 곳을 찾는다. 그리고 그 뒤에 이 칸을 채우는 일을 하는데, for 바로 다음부터 프로그래머가 실수를 한 것으로 보인다. for를 돌면서 minor 변수는 0부터 ACM_TTY_MINORS − 1까지의 값을 가질 수 있다. 이 구간에 속한 minor 값으로 ACM_TTY_MINORS 크기의 acm_table 배열을 접근하는 것은 올바른 코드이므로 아이락은 for의 조건에 대해서는 경보를 내지 않는다다. 그러나 for가 끝난 뒤부터는 minor가 ACM_TTY_MINORS의 값을 갖거나 minor가 $[0, \texttt{ACM_TTY_MINORS}]$의 값을 갖고 acm_table[minor]가 0을 갖는다. 아마도 이 부분을 짠 프로그래머는 for가 끝난 뒤에도 acm_table[minor] 칸이 0이 아니라면 원하는 minor 값을 찾지 못한 것이므로 오류 값을 돌려주려는 의도였을 것이다. 그러나 acm_table에서 값이 0인 칸을 찾

지 못한다면 minor는 무조건 ACM_TTY_MINORS라는 값을 가지며 이러한 인덱스로 acm_table을 참조하면 안 된다는 사실을 간과한 모양이다. 아이락은 $[0, \texttt{ACM_TTY_MINORS}]$의 값을 갖는 minor로 크기가 ACM_TTY_MINORS인 acm_table 배열을 접근할 수 있다고 for 다음에 나온 모든 관련 코드에 대해서 경보를 낸다.

5.2 GNU tar의 rmt.c

앞서 설명한 Linux kernel의 예와 비슷한 경우다. GNU tar 1.13의 src/rmt.c의 125번째 줄에 다음과 같은 get_string 함수가 있다.

```
main (int argc, char *const *argv)
{
...
        char device_string[STRING_SIZE];
        get_string (device_string);
...
}

static void
get_string (char *string)
{
  int counter;

  for (counter = 0; counter < STRING_SIZE; counter++)
    {
      if (safe_read (STDIN_FILENO, string + counter, 1) != 1)
        exit (EXIT_SUCCESS);

      if (string[counter] == '\n')
        break;
    }
  string[counter] = '\0';
}
```

main 함수의 여러 군데에서 get_string 함수에 크기가 STRING_SIZE인 배열을 넘겨주어 채우는 작업을 한다. 여기서 get_string의 counter라는 변수가 갖는 값의 범위를 살펴보자. counter는 0부터 시작해서 for를 돌면서 STRING_SIZE까지 값이 증가한다. for 안에서도 string을 counter를 가지고 참조하지만 $[0, \texttt{STRING_SIZE} - 1]$의 값으로 STRING_SIZE인 배열을 접근하는 것이기 때문에 아이락은 문제가 없다고 판단한다. 그런데 for가 끝나고 그 아래로 진행을 할 때에는 counter의 값이 STRING_SIZE가 될 수 있다. for 안에서 string[counter]의 값이 '\n'이었다면 break로 빠져나오므로 STRING_SIZE보다 작은 값이겠지만, '\n'을 발견하지 못했다면 counter 값이 STRING_SIZE인 상태로 string 배열을 접근해 '\0'을 대입하게 된다. 즉, 크기가 STRING_SIZE인 메모리 영역을 $[0, \texttt{STRING_SIZE}]$의 인덱스로 접근하므로 아이락은 이 코드에 문제가 있다고 경보

를 낸다. 만약 STRING_SIZE보다 작은 크기의 배열을 get_string으로 넘기는 코드도 있었다면 아이락은 이 역시 빠뜨리지 않고 찾아냈을 것이다.

5.3 빠뜨린 예외 처리

다음은 실제로 제품에 사용되었던 어떤 임베디드 소프트웨어의 일부분이다.

```
static EVENTQUEUE       *_inputQueue[MAX_NUM_QUEUE];
...
int ew_Os_CheckInputEvent (ULONG qid, EVENT *pEvent) {
        int curQ = ew_GetQueue( qid );
        if( curQ < 0 )
                return ERR_FATAL;
        if( _inputQueue[curQ]->getPos >= DF_MAX_EVENT_ITEM )
                return ERR_FATAL;
...
}

int ew_Os_UpdatePos (ULONG qid, int type, int x, int y) {
        int curQ;
        curQ = ew_GetQueue( qid );
        if((_inputQueue[curQ]->eventItem[_inputQueue[curQ]->getPos]).type != type)
                return ERR_FATAL;
...
}

int ew_GetQueue( unsigned long qID ) {
        int i = qID - QID_SEED;
        if( _inputQueue[i] == NULL_Q )  return -1;
        if( i >= MAX_NUM_QUEUE || i < 0 ) return -1;
        return  i;
}
```

ew_GetQueue라는 함수는 배열의 인덱스를 돌려주는 함수인데, 어떤 상황에는 예외를 의미하는 -1을 돌려주기도 한다. ew_GetQueue를 사용할 때에는 프로그래머가 예외의 경우인지 먼저 검사한 뒤에 정상적인 경우에만 배열 참조에 그 결과를 사용해야하는데, 이를 실수로 빠뜨리고 프로그램을 짜는 경우가 생긴다. 아이락은 위의 ew_GetQueue라는 함수가 $[-1, \texttt{MAX_NUM_QUEUE} - 1]$의 값을 돌려준다는 것과 _inputQueue 배열의 크기가 MAX_NUM_QUEUE라는 사실을 분석해낸다. ew_Os_CheckInputEvent 함수 안에서 ew_GetQueue의 결과를 담은 curQ가 0보다 작은 경우에는 함수를 끝내므로, 그 아래를 진행할 때에는 curQ가 $[0, \texttt{MAX_NUM_QUEUE} - 1]$의 값을 가진다는 사실을 안다. 따라서 curQ를 _inputQueue의 인덱스로 사용하는 것에는 아무런 문제가 없으므로 아이락은 여기서 경보를 내지 않는다. 그러나 ew_Os_UpdatePos 함수에서는 ew_GetQueue가 돌려준 $[-1, \texttt{MAX_NUM_QUEUE} - 1]$의 값 그대로 _inputQueue를 참조할 수 있기 때문에 아이락이 오류가 발

생할 수 있다고 경보를 내며 배열의 크기와 인덱스 값을 알려준다. 아이락은 이렇게 프로그래머가 실수로 예외처리를 빠뜨린 경우를 정확히 찾아준다.

앞에서 소스코드와 함께 소개한 분석 결과를 살펴보면 프로그래머가 만들어내는 오류들은 그 형태가 의외로 간단하다. 그러나 이렇게 간단한 구조의 오류들이라도 전체 프로그램 코드가 매우 큰 경우에는 사람의 손으로 찾아내기가 결코 쉽지 않다. 프로그래밍이란 아주 작은 실수에도 깨질 수 있는 매우 섬세한 조각들을 서로 정교하게 이어 붙여서 원하는 일을 하는 구조물을 만드는 작업이다. 어느 한 부분이 살짝이라도 어긋나 있다면 구조물 전체를 와르르 무너뜨리는 것이 바로 프로그램의 버그다. 해야 할 일이 명확하며 이를 비교적 체계적으로 구현한 프로그램에서 어긋난 부분들이란, 앞서 살펴본 예들과 같이, 보통 사소한 실수에서 비롯되기 마련이다. 사소한 실수들이지만, 커다란 프로그램의 구석구석에 숨어있다면, 이를 찾아내는 데에는 그 구조물을 만들어내는 것보다도 더 큰 노력이 필요할 수 있다. 아이락은 이렇듯 사소한 실수로부터 비롯되는 치명적인 문제들을 자동으로, 빠뜨림 없이, 그리고 실행 전에 찾아준다.

6 분석 속도와 정확도

GNU 프로그램들, Linux kernel 및 상용 임베디드 소프트웨어를 대상으로 한 아이락의 분석 결과는 다음 표와 같다. Pentium4 3.2GHz CPU와 4GB 메모리를 가진 시스템에서 아이락을 돌린 시간이다. 줄 수는 전처리하기 전의 C 프로그램의 줄 수이다. 경보 수에서 배열은 경보를 낸 대상 메모리 영역 또는 배열의 수이며, 접근은 그 대상들을 잘못된 인덱스로 참조한 위치의 모두 센 것이다.

위의 결과에서의 분석 속도는 초당 평균 10줄 정도로 컴파일 시간 등과 비교해볼 때 별로 빠르지 못하다. 그러나 6000줄짜리가 1초도 안 되서 끝나는가 하면, 1200줄짜리가 45초가 넘게 걸리는 등 프로그램이 하는 일에 따라서 분석 속도의 편차가 심하다. 물론 개선의 여지는 항상 남아있지만, 아이락이 모든 경우를 다 포함하는 결과를 만들어준다는 사실을 생각해보면 테스트를 통한 검증과는 비교가 안될 만큼 빠른 속도라고 생각할 수도 있겠다.

위의 결과에서의 분석 정확도는, 찾아낸 총 1199개의 접근 오류 중에서 199개가 실제 오류로 16.6%이다. 경보가 나오면 이들이 실제 오류인지 아닌지는 사람이 직접 그 진위 여부를 확인해야 한다. 가령, Es* 소프트웨어의 경우를 보자. 162개의 경보 중에서 겨우 6개의 실제 오류를 발견하기 위해서 나머지 156군데를 살펴봐야하는 헛수고를 해야만 한다고 생각할 수 있다. 그러나 23만 줄의 프로그램 코드를 전체를 살펴보는 대신에 아이락이 짚어준 162 군데만 살펴보면 된다. 실제로 발생할 수 있는 모든 잘못된 배열 참조 문제를 이 162군데만 확인함으로써 찾아낼 수 있다는 것이다. 정확도를 단순히 실제 오류와 경보 수의 비로만 생각할 것이 아니라, 분석한 프로그램의 크기까지도 고려하여 그 효용성을 판단하여야 할 것이다.

7 아이락의 한계

아이락 같은 안전한 프로그램 분석기에서는 분석속도와 정확도간의 밸런스가 상당히 중요하다. 대다수의 경우에 정확도를 높이려다 보면 분석시간이 늘어나고 분석속도를 높이려다 보면 정확도가 떨어진다. 아이락의 경우 앞에서의 실험결과를 보면 알 수 있듯이 사용자가 프로그램을 분석하여 바로바로 분석결과를 확인하고 이를 바탕으로 디버깅을 할 수 있는 수준의 분석속도를 갖고 있지는 못하다. 하지만, 테스팅을 통해 디버깅하는 과정과 비교해 생각해보면 이 정도의 시

종류		줄 수	시간(초)	경보 수		실제 오류 수
				배열 수	접근 수	
GNU 소프트 웨어	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
상용 임베디드 소프트웨어	T*	109,878	4525.02s	16	64	1
	U*	17,885	463.60s	8	18	9
	S*	3,254	5.94s	17	57	0
	SD*	29,972	457.38s	10	140	112
	DD*	19,263	8912.86s	7	100	3
	DC*	36,731	43.65s	11	48	4
	DS*	138,305	38328.88s	34	147	47
	Es*	233,536	4285.13s	28	162	6
	EI*	47,268	2458.03s	25	273	1

간 안에 모든 버그를 빠뜨림 없이 찾아낸다는 것은 유용할 수 있다. 안전한 프로그램 분석기의 분석결과에는 허위 경보(false alarm)가 들어가지 않을 수가 없다. 허위 경보는 실제로 버그가 아닌 곳을 버그라고 하는 것이다. 안전하게 모든 것을 포섭해서 분석을 하려다 보면 요약하는 과정을 피할 수 없고, 그런 요약은 분석의 정확도를 떨어뜨릴 수밖에 없다. 다음의 코드를 보자.

```
s=0;
for(i=0;i<bound;i++){
   s += i;
}
```

이렇게 간단한 프로그램을 분석한다고 했을 때 변수 `i`가 변수 `bound`가 가지는 값이라는 사실은 정확히 분석할 수 있지만, 변수 `s`가 0부터 (변수 `bound`가 가지는 값 - 1)까지의 합이라는 것을 정확히 분석하기란 여간 힘든 것이 아니다. 요약해석(abstract interpretation)의 결과로는 `s`가 0이

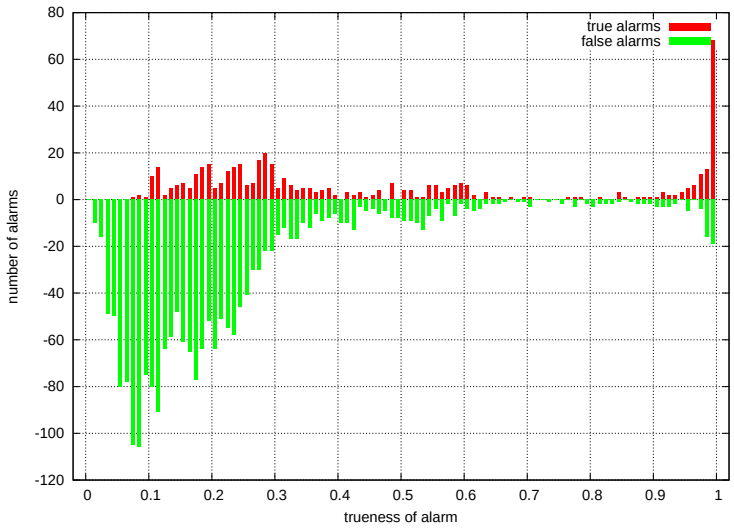

그림 1: 경보들의 참일 확률 분포를 보여준다.

상의 수라는 사실 밖에는 알아내지 못한다. 물론 더 구체적인 수를 알아내기 위해서는 더 많은 분석 시간을 사용하면 된다. 하지만, 만약 `bound`가 프로그램의 외부 입력과 같이 모르는 값이라면 어떻게 해야 할까? 안전한 분석을 위해서는 그 모르는 값이 모든 정수라는 가정 하에 분석을 해야 한다. 모든 정수를 실제 정수들로 표현하면 무한하기 때문에 요약 없이는 분석이 유한한 시간에 끝날 수 없다. 요약은 이렇게 필수적이고 그래서 허위경보들이 나오는 것이다. 그런데 너무 많은 허위 경보는 분석기의 사용자를 지치게 한다. 경보가 난 몇 군데를 계속 살펴봤는데 모두 실제로는 버그가 아니었다고 판단이 든다면 사용자는 분석기를 못 믿게 된다. 이런 허위 경보를 줄일 수 있는 하나의 방법으로 우리는 베이지안 통계 분석(Bayesian statistical analysis)을 사용했다. 스팸메일(spam mail)을 걸러내는 방법과 유사한 방법이다. 실제 스팸메일인 경우들을 어느 정도 알고 있는 상태에서 메일을 받았을 때 그 것이 가지고 있는 여러 가지의 증상(symptom)들을 보고 실제로 스팸메일일 확률을 계산하여 스팸메일을 분류할 수 있다. 우리도 실제 버그인 곳들을 알고 있는 상태에서 어떤 경보가 나왔을 때 그곳이 가지고 있는 여러 가지의 증상들을 보고 실제 버그일 확률들을 통계적으로 계산해 낼 수 있다. 여기서 증상이라는 것은 경보가 나온 곳이 가지고 있는 문법 구조, 실행 흐름, 분석 중에 나타나는 성질 등 이외에도 특성으로 삼을 수 있는 모든 것들이 될 수 있다. 이 방법의 장점은 분석결과로 나오는 경보들을 실제 버그들과 허위경보들로 분류한 정보를 많이 제공하면 할수록 다음 예측이 정확해진다는 점이다. 그림 1의 그래프는 이 방법을 통해 아이락의 분석결과들이 실제 버그일 확률에 따라 실제 버그들과 허위 경보들의 개수를 그린 것이다.

허위 경보의 분포만을 보면 비교적 바람직한(실제 버그일 확률이 낮을수록 많아지는) 모습을 보여준다. 실제 버그의 경우에는 안타깝게도 확률에 따라 분포하는 모습을 보이지 못하는데, 이는 실제버그에 대한 정보가 적고 허위경보에 대한 정보만 많기 때문일 거라고 추측하고 있다. 사실 실제 프로그램들에서 실제 버그들을 많이 찾아내기란 쉬운 일이 아니다. 모두들 버그 없는 프로그램들을 짜려고 노력하고, 또 오픈 소스들에서는 아직 개발 중인 소스보다 버그가 더욱 드물게 발견된다. 이렇게 실제 버그일 확률을 계산함으로써 우리는 사용자에게 어떤 버그들을 먼저 보여줄 지를 랭킹을 정할 수 있다. 실제 버그일 확률이 높은 경보들을 우선적으로 보여줌으로써

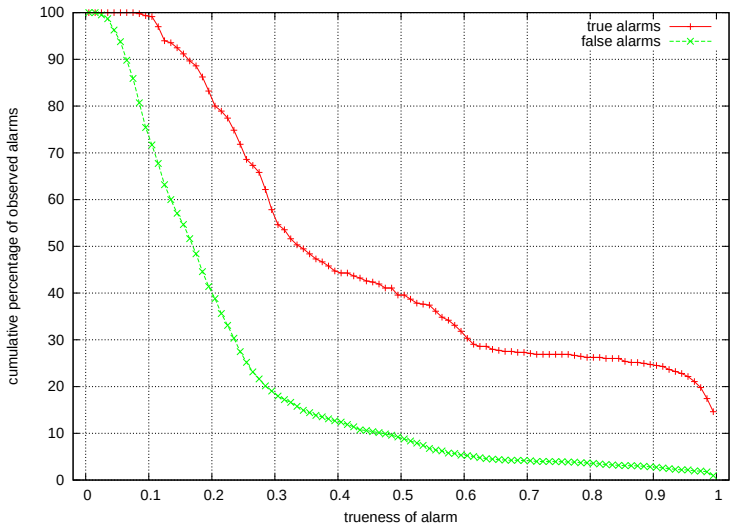

그림 2: 경보들을 확률이 높은 순으로 확인해 나갔을 때 발견하는 실제 오류와 허위 경보의 수를 보여준다.

사용자들이 허위경보에 지치는 것을 최대한 막을 수 있다. 그림 2의 그래프는 실제 버그일 확률에 따라 실제 버그와 허위경보들의 누적개수를 보여준다.

8 앞으로 할 일

일반적으로 프로그램 분석기란 프로그램이 가질 수 있는 성질을 알아내는 프로그램이다. 그 프로그램 성질 중에 아이락이 관심이 있는 성질은 "C 프로그램에서 스택이나 힙에 할당될 수 있는 메모리가 항상 처음에 할당되었던 크기안에서만 사용되는가?"이다. 명확히 정의 될 수 있는 프로그램의 성질은 모두 분석기의 분석 대상이다. 예를 들어 "이 프로그램은 항상 끝나는가?", "동적으로 할당되었던 메모리들은 프로그램이 끝나기 전에 모두 해제(free)되는가?", "이 프로그램이 실행중에 사용하는 메모리의 최대 크기는 어느 정도일까?"등등이 프로그램의 성질이 될 수 있다. 따라서, 이런 프로그램 성질을 알아내는 분석기는 아이락에서와 마찬가지로 요약해석(abstract interpretation) 방법을 통해 모두 구현 가능하다. 모든 프로그래밍언어는 그 언어로 짜여진 모든 프로그램의 의미를 정확히 정의할 수 있기 때문에 모두 분석대상언어가 될 수 있다. 즉 앞에서 소개된 C' 언어로 변환하는 과정만 구현 한다면 C가 아닌 다른 그 어떤 언어도 아이락의 분석대상언어가 될 수 있다. 정확도와 분석 속도도 여러가지 기술들을 이용하여 개선이 가능하다. 우리 연구실에서는 다양한 프로그램 성질들을 알아낼 수 있는 분석기를 만드는 연구를 하고 있다. 이와 동시에 분석대상이 되는 언어도 확장하고, 분석속도와 정확도를 높이는 방법들에 대한 연구도 우리 연구실에서 하고 있다.

참고 자료

[1] Airac. C 프로그램 배열 인덱스 분석기 설계에 대한 소개. `http://ropas.snu.ac.kr/airac/`.

[2] Astree. `http://www.astree.ens.fr`.

[3] Bandera. http://bandera.projects.cis.ksu.edu/.

[4] SLAM. http://research.microsoft.com/slam/.

[5] Kwangkeun Yi. 프로그램 분석 4541.664A 강의. 프로그래밍 언어의 기본적인 이론 및 프로그램 분석에 대한 소개. http://ropas.snu.ac.kr/~kwang/4541.664A/05/. 서울대학교 컴퓨터 공학부, 2005.