정확한 의미 전달을 위한 안전한 프로그래밍 언어

서울대학교 전기공학부
안준환

2010년 10월 11일

1 서론

최초의 범용 컴퓨터로 알려진 에니악(ENIAC; Electronic Numerical Integrator And Computer)이 개발된 1946년 이래로 컴퓨터는 시뮬레이션과 같은 복잡한 계산을 수행하는 용도부터 동영상 감상 등의 흥미를 위한 목적까지 특정한 용도에 국한되지 않고 널리 사용되었다. 이렇게 컴퓨터가 '범용적'이라는 특성을 가졌다는 장점은 컴퓨터의 발전이 사회 전반에 영향을 주게 되는 중요한 요소가 되었다.

프로그래밍 언어는 이렇게 다양한 목적으로 활용될 수 있는 컴퓨터에 인간이 원하고자 하는 바를 전달하기 위해서 탄생하였다. 현대의 컴퓨터 구조에서 프로그램을 실행하는 과정은 대부분 중앙 처리 장치가 명령어를 해석해서 기계의 상태를 변화시키는 작업이다. 인간과 기계 사이에 위치해 있는 프로그래밍 언어는 이러한 처리 과정을 지시할 수 있는 방법을 제공한다고 할 수 있다.

이 글에서는 과거와 현재의 프로그래밍 언어에 대해 알아보고, 미래의 프로그래밍 언어는 어떤 목적을 가지고 발전할 것인지에 대해서 논할 것이다.

2 본론

2.1 기계 중심의 언어

가장 널리 사용되는 언어 중 하나인 C 언어의 배경에는 B 언어가 있다. 이 언어는 1960년대 말, 켄 톰슨(Ken Thompson)에 의해 DEC PDP-7에서 수행되는 운영 체제 유닉스(Unix)를 개발하는 과정에서 정의된 시스템 프로그래밍

언어로, BCPL이라는 간단한 언어에 기반을 두고 있다. 이 언어의 컴파일러는 B 언어로 작성된 프로그램을 몇 가지 기본적인 동작을 수행하는 명령어의 주소값의 집합으로 번역하는 역할을 하였다. 따라서 B 언어로 작성한 프로그램의 수행 성능은 기계어로 작성된 프로그램에 비해 현저히 떨어졌음에도, 기계어로 프로그램을 작성하는 어려움 때문에 B 언어는 점점 더 널리 이용되었다.

그러나 PDP-11이 등장하면서 비효율적인 문자·포인터 처리, 부동 소수점 미지원 등 B 언어의 문제점이 지적되었다. 이러한 문제를 보완하기 위해서 데니스 리치(Dennis M. Ritchie)는 B 언어를 확장해 NB 언어를 정의했으며, 여기에 구조체 등의 타입과 여러 가지 연산자 등을 추가한 것이 C 언어의 시작이다. [1]

이와 같이 BCPL에서 C 언어까지 프로그래밍 언어가 발전한 역사의 중심에는 기계가 있었다. 즉, 사람이 기계에게 지시를 내리는 관점이 아니라 기계가 수행하는 관점에서 발달된 프로그래밍 언어였다. 이 때문에 C 언어는 기계의 많은 부분을 사용자에게 노출하고 있다. 사용자가 직접 사용할 메모리를 할당하고, 다 쓴 메모리를 해제해야 한다는 것, 다음으로 실행할 명령을 명시하는 `goto`가 언어의 일부로 남아 있는 것 등이 대표적인 예이다. 이러한 특성으로 인해 사용자의 작은 실수는 프로그램 전체가 중단되는 원인이 되기도 하고, 이를 찾는 것 또한 쉽지 않게 되었다.

C 언어를 기반으로 발달한 C++ 또한 크게 다르지 않다. C 언어에 객체 지향 프로그래밍(OOP), 참조, 템플릿 등의 특성을 추가하면서 보다 사람이 이해하기 쉬운 프로그래밍 언어를 목표로 발전한 것이 C++이지만, 메모리 관리나 약한 타입 시스템 등 C 언어의 기계적 특성이라고 지적되는 부분은 하위 호환성을 이유로 크게 개선되지 않았다. 이는 BCPL에서 C 언어가 발전한 것처럼 기존 언어의 근본적인 문제점을 해결하는 것이 아니라 그 때 그 때 필요한 기능을 추가해서 발전해 온 언어이기 때문이다.

이러한 기계적 특성을 사용자가 직접 다루는 것은 프로그램의 안전성에 큰 위협이 된다. 이는 근본적으로 기계 중심의 프로그래밍 언어가 프로그램에서 오류를 발견하는 것을 대부분 사용자에게 위임하고 있기 때문이다. 실제로 많은 프로그래머들이 프로그램을 작성하는 시간보다 프로그램에 존재하는 오류를 찾는 데에 더 많은 시간을 소비하며, 이는 생산성 저하와 잠재적인 오류에 대한 불안감으로 이어지게 된다.

따라서 보다 안전한 프로그램을 작성할 수 있도록 프로그래밍 언어 수준에서 돕는 방법이 필요하다는 결론에 이르게 된다.

2.2 논리학을 기반으로 한 프로그래밍 언어

논리학은 고대 아리스토텔레스 이래로 꾸준히 발전했으나, 현대 논리학의 시작은 고틀로프 프레게가 저술한 개념표기법(Begriffsschrift)이라고 볼 수 있다. 독일의 논리학자 게하르트 겐첸(Gehard Gentzen)은 이 책에서 그가 사용한 표기법과 공리를 기반으로 하는 체계를 수정해서 몇 가지 직관적인 규칙을 기반으로 증명하는 자연 연역법(natural deduction)을 완성하였다. 또한 그는 $\Gamma \vdash A$에 대한 증명은 항상 Γ, A 또는 이들의 부분만으로 표현할 수 있다는 부분식 성질(subfomula property)을 증명하였다.

한편, 미국의 논리학자 알론조 처치(Alonzo Church)는 논리식을 표현하는 또 다른 방법으로 람다 대수(lambda calculus)를 제안한다. 이 방법의 특징 중 하나는 함수에 값을 대입하는 과정을 단순화한 것이다. 예를 들어 $f(x, y) = x \times y$라는 함수에 $x = 3$, $y = 4$를 대입하는 과정은 식 1과 같이 표현된다.[1]

$$((\lambda x.(\lambda y.x \times y))(3))(4) \Rightarrow (\lambda y.3 \times y)(4) \Rightarrow 3 \times 4 \Rightarrow 12 \tag{1}$$

이후에 그는 튜링과 함께 튜링 기계에서 계산되는 모든 함수를 람다 대수로 나타낼 수 있다는 사실을 증명하였고, 기존의 람다 대수에 타입을 추가한 단순 타입 람다 대수를 제안하였다. 이는 람다 대수를 기반으로 하는 함수형 프로그래밍 언어의 이론적 기반이 되었다.

서로 다른 것처럼 느껴지는 이 두 체계가 사실 동등하다는 것을 발견한 것은 해스켈 커리과 윌리엄 하워드이다. 이러한 커리-하워드 대응 관계에 의하면 람다 대수 표현에 대입하는 것을 동등한 증명 과정으로 바꿀 수 있기 때문에 프로그램과 증명은 동등하다는 결론을 얻게 된다. [3]

이러한 결론은 프로그램의 안전성을 보장하는 데에 큰 도움을 줄 수 있다. 어떤 일을 하는 함수를 프로그래밍하는 대신, 어떠한 가정에 대해 원하는 결론을 얻을 수 있음을 이론적으로 보이고, 이를 대응되는 프로그램으로 번역하면 되기 때문이다. 이는 C 언어나 C++와 같은 기계 중심의 언어를 사용할 때 기계가 해야 할 일을 나열하는 방법으로 프로그램을 작성했던 것과는 근본적으로 다른 방식이다.

앞에서 논의한 패러다임을 반영한 실제 프로그래밍 언어 중 하나로 Objective Caml을 들 수 있다. 로빈 밀너(Robin Milner)는 람다 대수에 기반하는 엄밀한 프로그래밍 언어인 ML이라는 언어를 설계하였는데, Objective Caml은 프랑스의 INRIA에서 설계한 이 언어의 방언(dialect)이다. 이 언어는 람다 대

[1] 람다 대수에서는 여러 개의 인자를 받는 함수를 반복적인 함수의 적용으로 표현한다. 이 아이디어는 해스켈 커리에 의해 커링(currying)이라고 명명된다.

수의 특징을 다수 반영하고 있는데, 대표적으로 함수를 일급 객체로 취급하는 것이나 커링을 지원하는 것 등을 들 수 있다. 이 외에도 안전한 프로그래밍을 위해 다음과 같은 다양한 기능을 제공한다.

힌들리-밀너 타입 추론 시스템 Objective Caml이 강한 타입 시스템을 가진 언어임에도 사용자에게 불편을 주지 않는 것은 타입을 일일히 명시하지 않아도 유추해 주는 타입 추론 시스템이 포함되어 있기 때문이다. 또한 타입이 올바르지 않게 사용된 프로그램은 이 시스템을 통과하지 못한다는 사실이 이론적으로 증명되어 있다. 이 덕분에 사용자의 실수를 방지하기 위한 강한 타입 시스템을 유지하면서도 불편을 최소화할 수 있다.

가비지 콜렉션 C 언어에서 사용자에게 메모리 관리를 전적으로 위임했던 것과는 달리, Objective Caml은 메모리를 할당하거나 사용하지 않는 메모리 공간을 해제하는 작업을 전담하는 기능이 포함되어 있다. 이 기능을 사용하면 프로그램이 수행하면서 잘못된 메모리 참조를 일으킬 수 없다는 것이 이론적으로 증명되어 있다.

패턴 매칭 `match` 구문을 사용하면 어떤 값이 가지는 패턴에 따라 서로 다른 동작을 수행하는 코드를 이해하기 쉽게 작성할 수 있다. 게다가 주어진 값이 가질 수 있는 모든 패턴을 명시했는지 확인한 뒤 사용자에게 어떤 패턴이 명시되지 않았다는 것을 알려 주기도 한다.

여기에 이르면 혹자는 프로그래밍 언어가 지나치게 이론적이면 학문적 용도 외에 실용성이 떨어질 것이라는 추측을 제기할 수도 있다. 그러나 이러한 부류의 언어가 산업계에서 상당히 유용하게 쓰이는 많은 사례가 존재한다. 한 예로 제인 스트리트 캐피탈이라는 회사가 Objective Caml이라는 프로그래밍 언어로 시스템을 개발한 것을 들 수 있다. 제인 스트리트 캐피탈이라는 회사는 금전을 다룬다는 특수한 환경으로 인해 프로그램에게 정확도, 기민성(agility), 고성능이라는 세 가지 특성이 요구되었다. 이 기업은 이러한 환경에서 강한 타입 시스템을 제공하는 Objective Caml을 사용해서 이해하기 쉽고 정확하며 효율적인 프로그램을 작성할 수 있었다고 한다. [5]

나 또한 강한 타입 시스템을 가지는 언어를 이용해서 프로그램을 작성하는 효율을 높인 경험이 있다. 몇 달 전 C 언어를 부가 효과(side effect)가 없는 표현식과 설탕 문법을 최소화한 추상 문법 트리로 바꾸는 프로그램을 Objective Caml로 작성한 적이 있다. 이 과정에서 C 언어와 같은 기계 지향적인 언어를 이용했을 때 어렵지 않게 만날 수 있는 다양한 메모리 오류 등을 걱정하지 않아도 되었던 것은 물론이고, 적절한 타입 정의를 사용한 결과 대부분의 실수

를 프로그램을 수행시키지 않아도 바로잡을 수 있었다. 또한 함수를 일급 객체로 취급하는 언어의 특성을 이용해서 코드를 보다 직관적으로 작성할 수 있었다. 예를 들면 리스트에서 특정 성질을 만족하는 원소만 거르는 작업을 할 때 Objective Caml의 `List.filter`를 사용하는 것이 C 언어의 반복문을 사용하는 것보다 짧고 이해하기 쉬운 코드를 작성하는 데에 도움이 되었다.

그렇다면 이렇게 서로 다른 두 개의 패러다임을 기반으로 발전한 프로그래밍 언어 중에 어떤 것이 더 좋다고 할 수 있을까?

2.3 좋은 프로그래밍 시스템

'좋다'는 단어는 개인의 주관에 상당히 의지하는 것이 당연하지만, 나는 좋은 프로그래밍 언어인지 판단하는 기준을 '정확한 의미 전달'과 '안전함'에 두고 싶다. 프로그래밍 언어의 초기 목적을 단순화하면 결국 기계와 인간의 소통이다. 소통에 있어서 가장 중요한 것은 내가 의도한 것을 상대방에게 정확하게 표현하는 것이다. 또한 내 의사 표현에 오류가 있으면 이를 바로잡아 주는 것도 필요하다.

이러한 목표를 달성하기 위한 노력은 크게 두 가지 방법으로 진행되고 있다. 하나는 앞서 알아본 것처럼 *프로그래밍 언어* 자체를 제대로 설계해서 올바르지 않은 프로그램 자체를 받아들이지 않는 것이다. 이러한 노력은 Haskell이나 Objective Caml과 같이 이론적 엄밀함을 기반으로 설계된 프로그래밍 언어를 통해 실현되고 있다. 다른 하나는 *프로그래밍 언어를 분석하는 방법*을 개선해서 기존의 언어를 안전하게 하는 것이다. C 언어로 작성된 프로그램에서 할당한 메모리 영역을 벗어나는 오류를 정적으로 분석하는 아이락(Airac)과 같은 프로그램에 적용된 요약 해석 방법이나 프로그램을 검증하는 데에 사용되는 고차 논리 증명기가 이와 같은 노력의 일부분이다.

앞에서 언급한 두 번째 방법은 첫 번째 방법과 동시에 사용될 수 있기 때문에 결국 좋은 프로그래밍 언어로 프로그램을 작성하는 것은 중요한 문제라는 결론에 도달한다. 그럼에도, 많은 사람들은 변화에 필요한 시간과 비용을 이유로 쉽게 새로운 언어를 시도하려 하지 않는데, 나는 진입 비용에 비해 새로운 프로그래밍 언어를 통해 얻을 수 있는 이익이 훨씬 더 크다고 생각한다.

3 결론

어떤 언어가 모든 경우에 대해 다른 언어보다 우월하다는 주장은 섣부른 주장일 수 있다. 다만 기계어와 같이 타입이 존재하지 않는 프로그래밍 언어부터

동적 타입 시스템을 가진 Python, 약한 정적 타입 시스템을 가진 C/C++ 언어, 강한 정적 타입 시스템을 가진 Objective Caml과 Haskell 등 다양한 종류의 언어를 경험해 본 입장에서 보면 강한 정적 타입 시스템과 그 이론적 기반은 프로그램의 오류를 발견하는 데에 큰 도움이 되는 것은 분명하다.

논리학에 기반한 프로그래밍 언어가 시사하는 점은 그 동안 컴퓨터가 하지 않았던, 혹은 할 수 없었던 일이 사실 컴퓨터가 해야 했던 일이었고, 논리학을 이용하면 보다 아름답게 할 수 있다는 것이다. 인간은 언제나 실수를 할 수 있기 때문에 보다 많은 것을 컴퓨터에 의존하게 될 것이며, 이러한 관점에서 보면 잘 정제된 프로그래밍 언어의 필요성은 점점 더 커질 것이다.

참고 문헌

[1] Dennis M. Ritchie, "The Development of the C Language," 1993.

[2] Karen A. Frankel, "An Interview with Robin Milner," *Communications of the ACM*, Vol. 36, 1993.

[3] Philip Wadler, "Proofs are Programs: 19th Century Logic and 21st Century Computing," 2000.

[4] Iwan Wand and Robin Milner, "Computing Tomorrow: Research Directions in Computer Science," Cambridge University Press, 1996.

[5] `http://ocaml.janestreet.com/?q=node/61`.