소프트웨어 기술의 발달과정과 nML의 역할

이광근
cs.kaist.ac.kr/~kwang
프로그램분석 시스템 연구단
전산학과
KAIST

[마이크로 소프트웨어], 2002년 7월호 [nML과 함께하는 프로그래밍 여행: 제 2 편]

nML 프로그래밍 시스템은 소프트웨어 기술의 발달과정에서 제 2세대 기술을 충실히 갖춘 언어이다. 소프트웨어 기술의 최종 목표가 무엇이고, 이 맥락에서 겨우 "제 2세대"라는 말이 무슨 얘기인지 살펴보도록 하자.

1 nML로 경험할 수 있는 프로그래밍 기술

nML로 경험할 수 있는 프로그래밍 시스템 기술중에서 C나 Java로는 겪어 볼 수 없는 것은, 프로그램의 오류를 자동으로 찾아내는 기술의 현재 수준이다. 소프트웨어의 오류를 자동으로 찾아주는 기술은 지금까지 2세대 기술이 완성되었는데, C와 Java는 1세대 기술만을 갖추고 있을 뿐이다. 1세대 오류 검증 기술은 1970년대에 달성된 것으로, 생긴게 잘못된 프로그램을 자동으로 찾아내는 기술이다. 이 기술이 문법검증기술(parsing)이고, 완전히 완성되어 어느 프로그래밍 언어에서나 제공되는 기술이다. 덕택에, 현재 누구도 문법 오류를 손으로 찾아내는 경우는 없다. C나 Java가 놓치고 있는 2세대 벅잡는 기술은 1990년대에 완성되기 시작한 기술인데, 타입 검증(type checking)이라는 기술이다. 이 기술은 프로그램이 실행중에 잘못된 값이 잘못된 계산과정에 휩쓸릴 수 있는 경우가 없는지를 컴파일러가 미리 안전하게 확인해 준다. C나 Java만을 사용하는 프로그래머들은 이렇게 미리 엄밀하게 확인해 주는 실용적인 기술이 가능하다는 사실을 알지

못한다. C나 Java만을 고집하는 프로그래머들은 불안해 해야 한다. 2세대 벅잡는 기술을 맛본 프로그래머는 그러한 기술의 가능성 마져도 알지 못하고 있는 프로그래머를 쉽게 능가할 수 있는 것은 뻔하기 때문이다. 사실 이제는 3세대 벅잡는 기술 – 생긴모습도 멀쩡하고, 실행중에 잘못된 값이 흘러들지도 않지만, 실행중에 가져야할 정교한 조건을 만족시킬 수 없는 프로그램을 컴파일러가 검증해 내는 기술 – 이 실험실 수준에서 성공을 거두고 있기까지 하다. 이렇게 꾸준히 발전해가는 기술의 핵심을 경험해 본 프로그래머들의 인구가 많아질수록 우리의 프로그래밍 기술수준이 글로벌 선두를 차지할 토양은 비옥해 질 것이다.

2 2세대 벅 잡는 기술은 아직 널리 퍼지지 못했다

아쉽게도 지금 가장 널리 쓰이는 시스템 프로그래밍 언어인 C와 Java등의 언어는 2세대 벅잡는 기술을 제대로 갖추지 못한 상태이다. "재대로 갖추지 못했다"는 것은 어느정도는 컴파일러가 검증해 주지만, 많은 경우 문제가 있는 데도 불구하고 컴파일러는 아무렇지도 않게 프로그램을 통과시킨다는 것이다. 실행되면서 문제가 발생하는 것을 방치하는 것이다. 다음의 C 프로그램의 예를 보자. 메모리 포인터(x)와 그 메모리 블락의 크기(size)를 받아서 그 블락을 정수 0으로 모두 초기화하는 함수이다:

```
int init(int *x, int size)
   int i;
   for (i=0;i<size;i++)
   *(x+i) = 0;
```

이제 다음과 같이 위의 함수 init 을 사용한다고 하자:

```
init(malloc(sizeof(char)*10), 10);
```

이 코드는 타입에 맞지도 않고 위험한 프로그램이다. 메모리 블락이 정수(int)들의 블락이어야 하는데 문자(char)블락이거니와, 대게 정수를 표현하는데 4 바이트를 사용하고, 문자를 표현하는데는 1 바이트를 사용하므로, 위와같이 문자 10개를 보관할 크기의 메모리 블락을 init 함수에 던져주면 그 함수는 정수크기의 보폭으로 10발짝 움직이면서 0으로 초기화 시킨다. 할당한 메모리(10bytes) 바깥의 부분까지(40bytes) 0으로 초기화 하는 일이 벌어지게 되는 것이다. 이러한 위험한 C 프로그램에 대해서 컴파일러는 타입검증을 제대로 못하고 경고만 줄 뿐 이다:

```
% gcc t.c
```

```
t.c: In function ‘alloc’:
t.c:11: warning: passing arg 1 of ‘init’ makes pointer from integer
without a cast
```

심지어는 프로그래머가 적절히 타입을 우기면(type cast) 위의 경고마저 불가능하게 할 수 있다:

```
init((int *)malloc(sizeof(char)*10), 10);
```

2세대 벅잡는 기술을 제대로 갖춘 언어라면 다음과 같아야 한다. 컴파일러가 주어진 프로그램을 검증해서 문제가 없다고 했다면, 정말로 실행중에 문제가 없다는 것이 보장되야 한다. 이미 이 기술을 효과적으로 구현한 실용적인 언어들이 속속 출현하고 있고, 그러한 부류의 언어들이 바로 ML 이나 Haskell 계열의 언어이며, 이 시리즈에서 소개할 nML이 그 한 예가 된다.

이 2세대 벅잡는 기술은 아직은 문법검증(parsing)기술 만큼 모든 프로그래머들이 늘상 사용하는 기술로 널리 퍼지지는 않았으나 그러한 분별있는 프로그래밍의 세계는 곧 펼쳐질 것으로 보인다. (이러한 언어들이 실용적으로 쓰이는 예들에 대한 궁금증은 다음의 웹페이지에서 부터 풀어볼 수 있을 것이다: `http://ropas.kaist.ac.kr/ kwang/functional-anger.html`) 2세대 벅잡는 기술이 프로그램 짜기를 얼마나 쉽게 해주는 지를 경험하려면 그러한 기술을 갖춘 언어로 프로그램을 해보는 수 밖에는 없다. 이와 관련해서 비록 작은 샘플이기는 하지만, 2세대 벅잡는 기술을 갖춘 nML 로 프로그램하면서 겪은 학생들의 이야기들이 `http://ropas.kaist.ac.kr/ kwang/320/02/essay/` 에 모아져 있다.

사실은 이 2세대 벅잡는 기술로도 아직은 미흡한 실정이다. 실행중에 모든 값이 분별있게 착착 흘러드는 프로그램이라고 해도, 생각대로 작동하지 않을 수 있기때문이다. 타입에 맞는다는 것은 실은 초보적인 조건일 뿐이다. 휘발유만이 비행기의 엔진에 흘러든다는 것이 검증되었다고 해도, 휘발유의 폭발력이 수준미달인 경우라면 비행기가 떨어질 수 있다. 라면을 끓이는 계산에 항상 라면과 물과 불이 들어선다고 해도, 물이 한방울 뿐이거나 들어선 불이 포항제철의 용광로정도라면 라면의 맛이 망쳐질 수 있다. 타입에 맞게 컴퓨터 프로그램이 실행되더라도 생각한 것과는 다르게 진행되는 경우, 이러한 버그를 자동으로 검증하는 기술이 필요하게된다.

제 3세대 벅잡는 기술은, 이렇게 확장된 버그를 검증하는 기술을 목표로한다. 생긴모습도 멀쩡하고, 실행중에 잘못된 값이 흘러들지도 않지만, 실행중에 가져야할 정교한 조건을 만족시킬 수 없는 프로그램, 이것을 집어내는 기술이다. 이 기술의 얼개는 다음과 같다. 프로그램이 실행중에 만족해야 하는 정밀한 속사정이 엄밀한 논리식으로 정의된다. 주어진 프로그램이 실행중에 그 논리식을 거짓으로 만들 수 있는 가능성이 조금이라도 있는지를 검증한다. 있으면, 그 프로그램은 벅이 있을 수 있는 것이고,

그 가능성이 전혀 없다고 판정되면 그 프로그램은 벅이 전혀 없는 것이다. "가능성이 조금이라도 있는지"라는 표현을 쓴 것은, 벅이 없는데도 불구하고 벅이 있다고 결론내리는 경우가 생기기 때문이다. 보수적인 것이다. 하지만 안전한 것은 보장된다. 벅이 없다고 결론내려지면, 정말로 없다는 것은 보장된다. 안전하기는 하지만 완전하지는 못한 검증이다. 하지만 우리는 완전하지 못하다는 것에 만족해야 할 것 같다. 완전한 검증이 불가능하다는 것은 증명된 사실이기 때문이다.

다시 위의 C 함수 init 의 예를 보자:

```
int init(int *x, int size)
   int i;
   for (i=0;i<size;i++)
   *(x+i) = 0;
```

이 함수가 제대로 작동하기 위한 조건은 단순히 타입이 맞아야 하는 것뿐 아니라, x가 가르키는 메모리 블락의 크기가 최소한 "정수크기×size"와 같아야 한다:

$$\text{block-size}(\texttt{x}) \geq \text{sizeof}(\text{int}) \times \texttt{size}$$

정수크기의 보폭으로 size만큼 움직이며 0으로 초기화 시키므로, 받아논 메모리 블락은 이러한 걸음거리를 모두 포함할 정도로 넓어야 한다. 이 조건을 확인하기위해선, 위의 함수가 호출되는 곳 마다, x 와 size 가 위 조건을 만족하는 지 확인해야한다. 만일에, x 가 가르키는 블락의 크기를 미리 아는 데 실패한다면, 위의 조건을 확인할 방법이 없으므로 벅이 있는 프로그램으로 간주해야 안전하다.

프로그램이 보증해야하는 성질을 검증할 때, "잘 모르겠는" 경우를 최소화 시키고, 그 검증이 항상 자동으로, 그리고 적은 비용으로 가능하도록 하는 기술, 이러한 3세대 벅잡는 기술이 성숙되어지는 시기는 낙관적으로 잡아서 앞으로 20년정도 이후가 아닐까 생각된다. 그때가 되면 많은 사람들이 쓰는 프로그래밍 언어들은 적어도 2세대 벅잡는 기술은 제대로 갖추어진 것들이 될 것이다. 지금 우리가 1세대 벅잡는 기술을 갖춘 언어를 당연한 것으로 생각하고 쓰고 있듯이.

3 nML의 기타 특징

이러한 기술 발전의 맥락에서, nML 프로그래밍 언어는 2세대 벅잡는 기술을 제대로 갖추고 있는 상태이고, 3세대 벅잡는 기술에 대한 연구결과를 실용적으로 담아낼 대상으로 준비해 놓은 상태라고 할 수 있다. 이런 언어로

프로그램을 작성하면서 얻을 수 있는 기타 장점을 정리해 보면 다음과 같다.

- 작고 간단하다: 문법이 작고 간단할 뿐만 아니라 그 의미구조(semantics) 또한 간결하다. 지난호에 소개한 "값중심의 언어"가 가지게 되는 성질일 것이다. 극명한 예로, ML의 참고서들이 200페이지에서 최대 400페이지를 넘지 않는 반면에 Java나 C++ 의 경우는 대부분 600페이지가 넘는다.

- 안전하다: 수행중인 프로그램의 완전한 안정성이 보장되어 있다. 즉, 컴파일러를 통과한 프로그램은 실행중에 파행적으로 중단되는 경우가 (core dump 나 segmentation fault등이) 발생하지 않는다. 이것은 언어 갖추고있는 강력한 타입 시스템 때문인데, ML(Haskell도 마찬가지)에서는 특히, 타입 검사를 타입 유추를 이용해서 자동으로 하기 때문에 프로그래머가 변수의 타입을 표기할 필요가 없다. 이 기술이 위에서 얘기한 "2세대 벅잡는 기술"이다.

- 타입에 얽매이지 않는 함수도 정의할 수 있다: 하나의 함수로, 타입은 다르지만 하는 일이 같은 함수들을 표현할 수 있기 때문에 프로그램하기가 경제적이고, 타입검사를 하는 언어가 가지는 경직성이 없다. 이것도 마찬가지로 "2세대 벅잡는 기술" 덕택이다. 타입 검증을 하는 Pascal이나 Java가 놓치는 기술이 되겠다.

- 편향된 프로그래밍 관성에서 벗어날 수 있다: 함수가 특이한 대상이 아니고, 여타 다른값과 구분 없이 (higher-order functions) 다루어 진다. C나 Java등의 언어에서 어느새 우리에게 스며든 프로그래밍 관성은, 함수는 특이해서 정수나 포인터등과 달리, 함수가 함수를 만들어 내거나 함수가 자유롭게 데이타로 취급되지 못한다. 함수를 저장하고 싶으면 함수 포인터를 써야하고, 포인터로 저장된 함수를 부르려면 특이하게 해야 한다.

- 명령형 프로그램도 가능하다: 메모리주소나 지정문 (assignment) 등으로 발생하는 메모리 반응들도 (side-effect들도) 아무 문제 없이 표현가능 하다.

- 정제된 모듈 처리기능을 갖추고 있다: 대형의 소프트웨어 개발에 꼭 필요한, 안전한 분리 컴파일 기능(separate compilation)이 제공되어 있다. 모듈들을 따로 컴파일해도 컴파일러가 타입오류를 안전하게 검증해 준다. 더군다나, 모듈을 일반화 시켜서 (모듈을 인자로 하는

모듈들을) 정의할 수 있기때문에, 하나의 모듈을 필요에 따라 구체화해서 재 사용할 수 있다.

또한, 대형 프로그램 개발의 필수 덕목인, 따로따로 포장하기(modularity)와 속내용을 신경쓰지않게하기(data abstraction) 등을 프로그래머가 제대로 잘 적용하고 있는지를 컴파일러가 검증해준다. 프로그래머가 모듈을 정의할 때 위의 덕목에 맞도록 정의하게 되는데, 그 모듈을 사용하는 파트에서 덕목에 어긋나게 사용되는 경우가 혹시 있는지 자동으로 검증해 준다. 이 검증도 nML이 가지고 있는 타입 시스템 덕택에 가능하게 된다.

- 자동으로 메모리를 관리해 준다: 효율적인 메모리 재활용 시스템이(garbage collection이) 제공되기 때문에 프로그래머가 메모리의 부족이나 재사용등에 대한 부담에서 자유롭다.

- 예외상황 관리(exception handling)이 간단하다: 이것을 이용 해서, 프로그램의 실행 중에 발생할 수 있는 긴급상황들을 (예를 들면, x/y 에서 y 값이 0인 경우들을) 정황에 맞게 프로그래머가 적절히 발생시키고 처리할 수 있다. 특히 nML에서는 발생하는 데 처리되지 못할 수 있는 예외상황이 어떤 것이 있는지 컴파일러가 자동으로 검증해 준다. Java에서와 같이 프로그래머가 함수마다 어떤 예외상황이 발생할 수 있는지 선언할 필요가 없다.

- 정형적인 의미구조를 갖추고 있다: 프로그램을 이해하는 데 필요한 정확한 정의가 제공되어 있다. 따라서, 프로그램의 의미에 혼동의 여지가 없으므로 프로그램을 분석하고 관리하는 작업이 용이하다.

- 프로그래밍의 미학을 익히기 쉽다: 대형의 고난도 프로그램이 드러내는 복잡성을 다룰 수 있는 프로그래밍 원리와 미학이 자연스럽게 표현된다. "값중심의 프로그래밍"을 지원하기 때문에 주장할 수 있는 덕목일 것이다.

지난호에 이어 이것으로 nML에 대한 오프닝 소개를 마치기로하자. 짧게 정리해 보면, nML로 프로그램을 짜면, 값중심의 간단한 생각으로 프로그램을 짜면서 실용적인 프로그램을 구축할 수 있을 뿐 아니라, 현재의 프로그래밍 기술수준(2세대 벅잡는 기술)의 제대로된 모습을 경험하게 될 것이라는 것이다.

필자는 지난 2년간 nML 프로그래밍 시스템을 이용해서 KAIST 학부 3학년 대상의 프로그래밍 언어 강의를 운영해 왔는데, 여기서 학생들의 소감을 있는그대로 전하면서 이번호 글을 마친다. 다음호 부터는 본격적인 nML프로그래밍 내용이 되겠다.

4 KAIST 2002년 전산과 3학년 대상 프로그래밍 언어 수강생들의 소감

- "여타언어로는 구현하기 힘든 것들이 nML로는 쉽게 구현되는 것을 보고 참 신기했던 기억이 납니다"
- "만약 어떤한 컴퓨터 랭귀지도 모르는 상태에서 nML이 자연스럽게 받아질 수 있을까 묻는 다면 Yes로 대답할 것이다. 다른 언어보다 쉽고 뛰어나다는 것은 아니다."
- "다른 과목 숙제, 프로젝트에서 C나 Java로 짜여 되는 걸 보면, '아 nML로 짜고 싶다'는 생각이 듭니다. nML프로그래밍이 지금은 확실히 편하고 생각하기도 편하구요."
- "nML 프로그래밍은 생각하는 방식이 너무 재미있습니다."
- "nML은 굉장히 인간중심, 개념 중심의 언어라고 느꼈습니다."
- "초기에는 당황하고 어색한 점이 많았습니다. 하지면 하면 할수록, 내가 생각하는 바를 그대로 프로그램으로 옮길 수 있다는 큰 장점을 발견할 수 있었습니다."
- "C나 C++등을 짜면서 생각이 점점 복잡하게 꼬여서 프로그램 짜는 데 애먹는 경우가 많았는데 nML은 간단하면서도 원하는 결과를 정확하게 얻어내서 프로그래밍 하기가 좋았습니다."

- “nML 프로그래밍 하면서 느꼈던 점 중에 가장 인상 깊었던 것은 일단 컴파일을 성공하고, 프로그램이 돌아갈 것이라 기대하고 샐행했을 때 정말 ’제대로’ 돌아가는 경우가 많았다는 점입니다. C++등의 언어는 컴파일 에러를 모두 없앤 후에, 런타임 에러를 없애는 데에도 많은 시간이 소비되지만, nML 프로그래밍에서는 일단 컴파일이 성공하면 대부분 내 생각대로 돌아가니까 처음에는 매우 신기했습니다.”
- “nML은 명령어가 많지 않아서 쉽게 알 수 있었습니다.”
- “nML을 하면서 정말 프로그램을 짜고 깔끔한 느낌이 든다는 것이 가장 인상적이네요. C++같은거에선 좀 느끼기 힘들었죠.”
- “슬슬 C로 짜는게 외 불편한지 느껴지기도 합니다.”
- “멋지다. 특히 디버깅 할때. 솔직히 디버깅 할 것도 없게 해주는 것 같지만.”
- “무엇보다도 nML이 우리의 언어라는 것이 자랑스럽습니다!”
- “nML을 처음 접했을 때 ’참 어렵다 그리고 이런 언어를 왜 쓰나’ 하는 생각을 했습니다. 하지만 시간이 조금씩 지나고, C++로 프로그램을 짜다보니 ’아 이거 nML로 쓰면 대개 편할텐데’라는 생각이 자주 들었습니다. nML이 좀 더 프로그래밍 언어에 대한 시각을 넓히는 데 도움을 주었습니다.”
- “뭔가 딱딱 맞아 들어가는 것이 지금까지와 다른 프로그래밍의 원리를 배울 수 있었습니다.”
- “퍼즐이 맞춰질 때의 그 쾌감같이 하나하나 내 생각되로 움직이는 게 재밌었습니다. nML이란 언어 참 편한거 같아요. 처음에는 익숙치 않아서 적응하기 힘들었는데 지금은 C나 Java보다 편리하다는 것이 느껴집니다. C에서는 너무 생각해야 될것이 많거든요. 어제 프로젝트 하는데 그놈의 세그폴 때문에.”
- “값중심의 프로그래밍. 처음에는 지금까지 흔히 쓰던 assignment가 되지 않아서 많이 힘들었습니다. 하지만 nML적 프로그래밍 방법이 훨씬 더 수학적 논리적으로 맞다는 사실을 알고서는, 아직은 부족하지만 nML이 정말 안정적인 언어라는 것을 알았습니다. type checking 또한 잘 되는 언어라는 점에서 상당히 안전한 언어입니다.”
- “C에 비해 월등히 사용하기 편리했고, 좋았습니다.”

- “아직도 nML프로그래밍에 그리 능숙하지는 않지만 처음 nML 프로그래밍을 했을 때에는 정말 생소해서 많이 어렵게 느껴졌습니다. 그리고 처음엔 C나 JAVA와 비교해서 이렇게 단순하게 생긴 언어로 충분히 복잡한 프로그래밍을 할 수 있을지 의문스러웠습니다. 그렇지만 지금은 오히려 C나 자바보다 더 간단하게 프로그래밍을 할 수 있을 것 같은 생각이 듭니다. conputer science를 공부하는 데 있어서 C나 JAVA보다 nML을 먼저 접한다면 더 도움이 될 것 같기도 합니다.”

- “매우 간단하고 깔끔하고 논리적인 언어였다. 프로그래밍의 아름다움을 느낄 수 있는 그런 언어였다.”

- “자료구조나 알고리즘을 그대로 표현할수 있다는게 인상깊었습니다. 다른언어를 써온 사람이라면 처음에는 기존의 언어에 익숙해져 있어서 nML 등의 언어의 장점, 유용함 등을 단번에 알기는 힘들것 같지만 익숙해지고 누구나 편리하다고 느낄만한 언어인것 같아요. 다른언어에 비해 매우 짧은 코드도 인상적이었습니다.”

- “nML은 불과 한달사이에 나를 마약처럼 사로잡아 버렸다. 메모리도 수많은 변수들도 신경쓸 필요가 없었다. 프로그래밍 내내 돌아다니는 것은 논리 그 이상도 이하도 아니었다. 생각하는 것을 바로 기호로 나타내고 컴퓨터는 방금 내가 했던 생각을 그대로 따라 원하는 결과값을 얻어낸다. 어떤것이 좋은 언어인지는 모르지만 잘 만들어진 언어라는 말은 바로 이런경우에 쓰는 것이라는 느낌이 들었다.”

- “처음에 nML로 수업을 진행한다고 했을 때 새로운 언어를 배워야 하는 부담감이 있었던 것이 사실입니다. 하지만 언어를 배워가면서 첫번째 숙제를 무난히 끝내고 나서는 오히려 nML이라는 언어가 이전의 언어보다 훨씬 재밌었고, 숙제가 기다려졌습니다. nML이라는 언어는 이전의 언어와는 다르게 프로그래밍이 쉬우면서도 그 기능은 강력합니다. 내가 생각하는 것을 표현하고 나서, 그 생각에 문제가 없다면 사사로운 실수등을 잘 잡아주어서 프로그래밍하기가 수월했습니다.”