□ 기술해설 □

Standard ML 프로그래밍

한국과학기술원[1] 이광근* · 류석영**

● 목 차 ●

1. 서 론

지금까지 대부분의 중요한 소프트웨어 – 많은 사람들이 오랫동안 사용하는 프로그램(운영체제, 컴파일러, 네트워크 시스템, 전화 스위칭 시스템 등) – 들이 C로 구현되어 왔지만, C 언어가 가지고 있는 문제점들과 최근 급속도로 축적되고 있는 프로그래밍 언어의 연구 성과를 볼 때, C가 앞으로도 계속해서 시스템 소프트웨어 개발의 주요 언어로 사용될 지는 의문이다[1,2,3,4,5].

물론, 이미 많은 연구자들이 C 보다 상위의 언어들의 장점에 대해 주장해왔고, 그러한 언어들로 실제적인 시스템 소프트웨어를 구성하려는 노력이 있었지만[2], 그러한 언어들은 C의 단점을 모두 해결할 수 없었고 배우기 어려울 뿐더러 성능도 뒤떨어지는 것이었다[6,7,9,11].

그러나 최근의 프로그래밍 언어에 대한 연구 성과들이 이러한 상황의 반전을 가능하게 해주고 있는데, 그 대표적인 예로 주목받고 있는 언어가 Standard ML(SML)[12,13,4,14,15]이다.

이 글에서는 SML을 C와 비교해서 살펴보기로 한다. SML을 소개하다보니 C에 대한 비판이 상대적으로 부각되는 경향이 있는데 대해 독자 여러분의 이해를 구한다.

2. 프로그램의 안전성

어떤 프로그램이 안전하다는 것은 그 프로그램이 실행 중에 잘못 다루어지는 값(run-time type error)이 발생하지 않는다는 것을 뜻한다[3]. 더욱이, 프로그램이 수행되기 전에 컴파일러에 의해 그 안전성이 검증될 수 있으면 그 프로그래밍 언어는 안전하다고 한다.

그렇다면, C 언어는 안전하다고 할 수 없다. C 컴파일러는 입력된 C 프로그램의 안전성을 확인해 줄 수 없기 때문에, 컴파일된 C 프로그램이 실행 중에 파행적으로 중단되는 경우가 흔히 발생한다. 예를 들면, 배열의 첨자나 메모리 주소로 엉뚱한 값들이 사용돼서 "bus

1) http://cs.kaist.ac.kr/

2) 예를 들어, Mesa를 이용한 Cedar system[6](Xerox PARC), Modula-2를 이용한 Topaz system[7,8](DEC Systems Research Center). CLU를 이용한 Swift system[9,10](MIT), Symbolics와 Xerox의 Lisp machines, Ada[11]를 이용한 실시간 내장 시스템 등.

*종신회원
**비 회 원

3) 프로그램이 프로그램 바깥의 환경(화일 시스템 등)을 위협할 수 있는 경우는 논외로 한다.

error"나 "segmentation fault"등으로 프로그램이 멈추게되는 경우 등이다.

반면에 SML은 수행중인 프로그램의 안전이 보장되어 있다. 즉, SML 컴파일러를 통과한 프로그램은 실행 중에 잘못되는 경우가 발생하지 않는다. 이것은 SML의 타입구조가 잘 정의되어 있어서 타입검사를 통과한 SML 프로그램이 실행 중에 잘못 처리되는 값이 발생하지 않는다는 것이 보장되기 때문이다. 더욱이, SML에서는 타입 검사를 타입 유추(type inference)를 이용해서 하기 때문에 프로그래머가 변수의 타입을 표기할 필요도 없다. 또한, 하나의 함수로 타입은 다르지만 하는 일이 같은 함수(복합형 함수 polymorphic function)를 정의할 수 있기 때문에 프로그램하기가 경제적이고, 타입검사를 하는 언어가 가지는 경직성이 적다. 이러한 복합형 함수가 정의되고 사용되면서도 프로그램의 안전성이 보장될 수 있도록 만든 타입구조를 복합형 타입구조(polymorphic type system)[16,17]라고 하는데, 여기서 한가지 언급해야 할 것은, 현재 SML이 가지고 있는 복합형 타입 구조가 완전하지는 못하다는 것이다. 다시 말해서, 안전한 SML 프로그램의 100%가 모두 타입 유추에 성공하는 것은 아니다. SML의 복합형 타입 구조가 고안됨으로써 보다 많은 프로그램이 SML 컴파일러를 통과할 수 있게는 되었지만(이런 의미에서 경직성이 적다는 것이지만), 안전한 SML 프로그램 모두가 컴파일러를 통과하는 것은 아니다.

3. 모듈 시스템

프로그래밍 언어가 실용적이기 위해서는 대형 소프트웨어의 각 부품이 개별적으로 개발되서 나중에 통합될 수 있는 방법이 제공되야 한다. 더군다나, 각 부품이 개별적으로 컴파일되고 최종적으로 통합(link)될 때 안전한 부품만이 최종 수행 코드를 구성할 수 있도록 보장되어야 한다.

C에서는 인터페이스 화일(.h file)과 내용 화일(.c file)이 한 쌍이 되어서 소프트웨어의 한 모듈을 이루고, 간단한 전처리기(text-based preprocessor)를 이용해서 개별 컴파일작업(separate compilation)이 이루어지지만, 각 모듈의 안전이 보장되지 못한다. C에서의 인터페이스 화일과 전처리기 방식만으로는, 외부 화일에서 정의된 함수의 타입이 정확히 확인될 수 없기 때문에(프로그래머의 의도나 실수로 쉽게 변환(cast)될 수 있는 등), 함수의 인자를 빠뜨리거나 순서가 바뀐 경우와 같은 간단한 타입 오류들이 컴파일 시간에 걸러지지 못한다. 이러한 것들은 C 프로그래머가 의도해서 편리하게 사용하는 경우도 있으나 대개는 심각한 프로그램의 오류로써, 실행 중에 파행적인 프로그램의 중단으로 나타나게 된다.

SML에서는 이러한 오류가 모두 컴파일 시간에 타입 유추를 통해 걸러질 수 있도록 모듈 시스템이 디자인되어 있다. 개별적으로 개발된 프로그램 모듈의 안전은 그 모듈이 컴파일될 때와 그 모듈을 사용하는 다른 모듈이 컴파일될 때 모두 검증된다. SML 에서는 C의 인터페이스 화일(.h file)에 해당하는 것을 시그너쳐(signature)라고 하고 내용 화일(.c file)에 해당하는 것을 스트럭쳐(structure)라고 한다. 더욱이, 모듈을 일반화시켜서(parameterized module) 정의할 수 있기 때문에, 하나의 모듈을 필요에 따라 구체화해서 재 사용할 수 있다. 이러한 일반화된 모듈을 펑터(functor)라고 하는데 C++의 "template class"나 Ada의 "generic package"를 상상하면 된다.

4. 프로그램의 정확한 의미

프로그래밍 언어의 의미가 애매모호하게 정의되면 많은 부작용이 있다. 프로그래머의 입장에서는 작성하고 있는 프로그램이 무슨 일을 하는지를 확실하게 알 수 없고, 컴파일러의 입장에서는 어떻게 돌아가도록 실행 코드를 만들어내야 하는지가 애매해진다. 결국은 컴파일러마다 다른 내용으로 실행 코드가 만들어질 수 있고, 프로그램의 이식성은 흔들리게 된다. 프로그램의 이식성뿐이 아니라, 더욱 심각하게는, 최적화 컴파일러의 작업이 어려워진다. 최적화할 프로그램의 내용이 정확하게 결정될 수 없으므로, 같은 내용을 가지는 최적의 실행 코드

를 만든다는 것이 애매해진다. 최적화 과정에서는 특히, 의미 구조의 정확한 정의뿐 아니라, 그 의미 구조가 작고 간단할 수록 바람직하다. 최적화된 실행 코드가 주어진 프로그램과 같다는 것이 보장되기 위해서는, 최적화 과정이 프로그램의 의미 구조를 깨뜨리지 않는다는 것이 보장되야 하는데, 이 때 의미 구조가 복잡하면 그 것을 확인하기가 난해해진다.[4] 대개의 C 컴파일러에서는 고급의 최적화 과정일수록 이러한 검증 과정을 거치고 구현되어 있지 않다. 때문에, 고급의 최적화 옵션을 사용해서 컴파일한 경우, 프로그램이 제대로 실행되지 않는 경우를 종종 맞닥뜨리게 된다.

C에서 대표적으로 애매한 경우는 포인터 타입을 변환할 때 발생한다. C에서는 타입의 변환이 자유롭게 이루어질 수 있는데, "큰" 값을 가지는 메모리 주소 타입의 변수가 "작은" 값을 가지는 메모리 주소 타입의 변수로 변환될 때의 정확한 의미가 정의되어있지 않다. 예를 들어, 캐릭터(char)를 기지는 주소 값(c)을 정수를 가지는 주소 값으로 변환시키는 것이 ((int *)c) 무슨 의미인지가 정의되어있지 않고, 그 반대의 경우만 정의되어있을 뿐이다[22, page 199 : 8-12].

반면에, SML은 의미 구조가 정확하게 정의되어 있을 뿐 아니라[12,13] 작고 간단하기 때문에, 배우기 쉽고[5] 프로그램의 의미에 혼동의 여지가 없으므로 프로그램을 분석하고 관리하는 작업이 용이하다. SML의 작고 간단한 면을 크고 복잡한 언어의 대명사격인 C++와 극명하게 비교해 보면, SML 참고서[12,13]가 총 261쪽이고, 초보자를 위한 SML 입문서[15]가 220쪽인 반면, C++의 경우는 각각 686쪽[23]과 464쪽[24]이다.

4) 안전한 최적화 과정을 디자인하는 틀로서, 요약 해석(abstract interpretation)[18,19], 집합 관계식(set-constraint)을 이용한 방법[20], 타입 유추(type inference)를 이용한 방법[21] 등이 있는데, 모두가 최적화 과정의 안전을 증명하는 데 프로그램의 의미 구조(semantics)를 이용한다.

5) KAIST의 컴파일러 과목은 SML을 이용해서 학생, 프로젝트가 진행되고, 성적은 각 팀의 컴파일러가 만들어내는 코드의 성능으로 결정되는데, SML을 처음 접하는 학부 학생들이 한 학기 동안 컴파일러의 전 과정을(약 4000줄) 구현하고 있다

5. 예외상황 처리(exception handling)

프로그램이 실행 중에 특별히 처리해야할 상황이 발생할 수 있다. 이러한 경우를 예외 상황(exception)이라 하고, SML에서는 프로그램 내에서 예외 상황을 정의하고, 발생시키고, 처리할 수 있다.

SML의 예외 상황 메카니즘은 예외 상황을 처리하는 데에만 사용되는 것은 아니다. SML에서는 프로그래머가 예외 상황을 발생시키는 지점에서 예외 상황이 처리되는 지점으로 프로그램의 실행 순서(control flow)를 급진적으로 바꾸고자할 때 유용하게 쓰인다. 예를 들어, 여러 겹으로 겹쳐져있는 함수 호출 상태에서, 곧바로 가장 바깥 함수가 호출된 지점으로 빠져나오도록 하는 데에 사용될 수 있다. C에서 longjmp와 setjmp를 이용한 점프(non-local goto)를 연상하면 된다. 예를 들어 다음 프로그램을 보자. 주어진 정수 리스트의 곱을 구하는 함수이다 :

```
exception ZERO and END
fun product(L) =
 let
  fun prod(nil) = 1 (* empty list *)
    | prod(head::tail) =
        if head = 0 then raise ZERO
          else head * prod(tail)
 in
  prod(L) handle ZERO => 0
 end
```

주어진 리스트 L의 곱을 구할 때(prod(L)) 원소 중에 0이 있을 때까지 prod는 여러 겹으로 재귀 호출되다가 0을 만나면 ZERO라는 예외 상황을 발생시킨다. 이때 중첩된 재귀 호출이 차례로 리턴되지 않고 곧바로 ZERO가 처리되는 장소인 초기의 prod 호출(prod(L))로 빠져나와서 0이 prod(L)의 결과가 된다.

6. 메모리 관리

프로그래머가 직접 메모리를 관리하면 치명

적인 오류를 가져오기 쉽다. 사용이 끝난 메모리를 재활용하지 않거나 너무 늦게 재활용하게 되면(메모리 출혈 memory leak) 프로그램이 메모리의 부족으로 중단될 수 있고, 반대로 메모리를 너무 빨리 재활용해 버리면(dangling pointer) 필요한 값을 잃어버리게 된다. 이러한 오류는 임의의 시간동안 잠복해 있다가 프로그램을 중단시키기 때문에 오류 수정이 매우 어렵다. 대형의 C프로그램이 가지는 오류는 대개가 메모리 관리의 잘못에 기인하는 경우가 많다.[6]

이러한 이유에서 많은 프로그래밍 언어가 자동으로 메모리를 재활용(garbage collection) 해주는 시스템을 제공하고 있고, SML도 예외는 아니다. 따라서 SML 프로그래머는 메모리의 부족이나 재사용 등에 대해서 신경쓰지않고 프로그램을 작성하게 된다.

7. SML 프로그래밍 예

7.1 이진 트리

SML에서는 프로그래머가 새로운 타입을 정의할 수 있다. SML의 데이터 타입은 재귀적으로(recursively) 정의된 데이터 타입을 비롯하여, 다양한 종류의 복잡한 데이터 구조를 간단하게 표현할 수 있다. 다음의 예는 정수를 가지는 이진 트리 데이터 타입을 정의하고 있다.

```
datatype tree = LEAF of int
              | NODE of tree * tree
```

위의 선언이 의미하는 것은, 프로그램에서 LEAF와 NODE는 각각 int와 두 개의 tree를 가지고 tree 타입의 원소를 만들어 내는 함수(SML에서는 데이터 컨스트럭터(data constructor)라고 한다)라는 것이다. 따라서 LEAF 5나 NODE(LEAF 1, LEAF2)가 모두 tree 타입의 값이 된다.

7.2 스택 모듈

SML 모듈은 스트럭쳐와 시그너쳐의 쌍으로 구성된다. 스트럭쳐는 데이타 구조와 그 데이타 구조를 사용하는 함수들을 하나의 묶음(encapsulating)으로 표현한다고 볼 수 있다. 시그너쳐는 스트럭쳐의 타입이다.

다음은 정수를 원소로 하는 스택 스트럭쳐이다:

```
structure Stack =
  struct
    exception EmptyStack
    type stack = int list
    val emptyStk = [] (* empty list *)
    fun push(x, s) = x::s
    fun pop(nil) = raise EmptyStack
      | pop(x::xs) = (s, xs)
  end
```

위에서 만든 스트럭쳐를 사용하여, 1만 가지고 있는 스택을 다음과 같이 만들 수 있다:

```
val myStack=Stack.push(1, Stack.emptyStk)
```

위의 스트럭쳐에 해당하는 시그너쳐는 다음과 같다:

```
signature StackSig =
  sig
    exception EmptyStack
    type stack
    val emptyStk: stack
    val push: int * stack -> stack
    val pop: stack -> int * stack
  end
```

이번에는 펑터를 사용하여 좀 더 일반화된 스택 스트럭쳐를 만들어내는 예이다. 스택의 원소 타입에 대해서 일반화시킬 수 있다. 다음의 펑터 StkFun는 타입 elmt를 정의한 스트럭쳐를 입력받아서, elmt를 원소로 하는 스택 스트럭쳐를 만들어낸다.

```
functor StkFun (S: sig type elmt end) =
  struct
    exception EmptyStack
    type stack = S.elmt list
    val emptyStk = []
    fun push(x, s) = x::s
    fun pop(nil)=raise EmptyStack
      | pop(x::xs) = (x,xs)
  end
```

6) 마이크로소프트의 Window 95의 출시가 연기되었던 이유가 메모리 출혈(memory leak)이었다고 한다.

따라서, 이 펑터를 사용하여 다음과 같이, 정수 스택이나 문자열 스택 스트럭처를 만들어낼 수 있다:

```
structure IntStk =
        StkFun (struct type elmt=int end)
structure StringStk=
        StkFun(struct type elmt=string end)
```

8. 결 론

이 글을 통해서 SML의 모든 면을 소개하지는 못했다. SML의 전체 모습에 관심있는 분들이 있으리라 믿으며, 그러한 독자 분들을 위해서 SML에 대한 자료들을 다음의 Web 페이지에 정리해 놓았다:

http://compiler.kaist.ac.kr/courses/sml.html

현재 SML 컴파일러는 거의 모든 Unix/MS Window 기계 위에서 작동하고 있고, 다양한 책자와 프로그래밍 도구들(yacc, lex, profiler 등)이 갖추어져있기 때문에, SML의 프로그래밍 세계를 경험하고자하는 데에는 어려움이 없으리라 믿는다.

참고문헌

[1] Niklaus Wirth. A plea for lean software, *IEEE Compuler*, 28(2):64-68, February 1995.

[2] T. Andrews, Designing linguistic interface to an object database or what do C++, SQL, and hell have in common? In *Proceedings of the 4th International Workshop on Database Programming Languages-Object Models and Languages*, Springer-Verlag, 1993.

[3] Guy L. Steele, Jr. Principled design of programming languages, *Invited presentation of ACM '94 Conference on Programming Language Design and Implementation*, 1994.

[4] Andrew W. Appel, A critique of Standard ML. *Journal of Functional Programming*, 3(4):391-430, 1993.

[5] Dennis M. Ritchie, The development of the C language, In *ACM SIGPLAN Notices, 28(3), History of Programming Languages Conference (HOPL-II)*, pages 201-208. ACM SIGPLAN, 1993.

[6] James G. Mitchell, William Maybury, and Richard Sweet, *Mesa language manual, version 5.0*, Rept. csl-79-3, Xerox PARC, 1979.

[7] Paul R. McJones and Garret F. Swart, Evolving the UNIX system interface to support multithreaded programs, Technical Report Rept. 21, DEC System Research Center, 1987.

[8] Paul Rovner, Extending Modula-2 to build large, integrated systems, *IEEE Software*, 3(6):46-57, 1986.

[9] David D. Clark, The structuring of systems using upcalls, In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 171-180, December 1985.

[10] Barbara Liskov, Russel Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder, CLU Reference Manual, In *Lecture Notes in Computer Science*, volume 114, Springer-Verlag, 1981.

[11] United States Department of Defense, *Reference Manual for the Ada Programming Language*, U.S.Government Printing Office, 1983.

[12] Robin Milner, Mads Tofte, and Robert Harper, *The Definition of Standard* ML, MIT Press, 1990.

[13] Robin Milner and Mads Tofte, *Commentary on Standard ML*, MIT Press, 1991.

[14] David B. MacQueen and Andrew W. Appel, Standard ML of New Jersey, Technical Memorandum, AT&T Bell Labs., 1993.

[15] Jeffery D. Ullman, Elements of *ML Programming*, Prentice-Hall, 1994.

[16] Luis Damas and Robin Milner, Principal type-scheme for functional programs, In *ACM Symposium on Principles of Programming Languages*, 1982.

[17] Robin Milner, A theory of type polymorphism in programming, *Journal of Computer and System Sciences*, 17:348-375, 1978.

[18] Patrick Cousot and Radhia Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, In *ACM Symposium on Principles of Programming Languages*, 1977.

[19] Patrick Cousot and Radhia Cousot, Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form, In *Lecture Notes in Computer Science*, volume 939, pages 293-308. *Proceedings of the 7th International Conference on Computer Aided Verification Edition*, 1995.

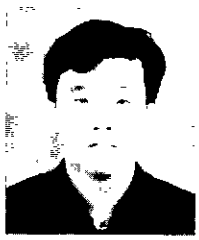
[20] Nevin Heintze, *Set Based Program Analysis*, PhD thesis, Carnegie Mellon University, October 1992.

[21] Jean-Pierre Talpin and Pierre Jouvelot, Polymorphic type, region and effect inference, *Journal of Functional Programming*, 2(3):245-271, July 1992.

[22] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall, 2nd edition. 1989.

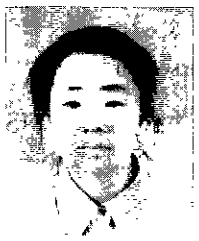
[23] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 2nd edition, 1993.

[24] Stanley B. Lippman, *C++ Primer*. Addison-Wesley, 1989.



이 광 근

1987 B.S. 계산통계학, 서울대학교 자연과학대학
1990 M.S. Computer Science, University of Illinois at Urbana-Champaign
1993 Ph. D. Computer Science, University of Illinois at Urbana-Champaign
1993~95 Member of Technical Staff, Software Principles Research Dept., AT&T Bell Laboratories
1995~현재 한국과학기술원 전산학과 조교수



류 석 영

1995 한국과학기술원 전산학과 학사(B.S.)
1996 한국과학기술원 전산학과 석사(M.S.)
1996~현재 한국과학기술원 전산학과 박사과정