# 프로그래밍 언어에서의 타입 구조의 역할*

이욱세 편기현 이광근†
한국과학기술원 전산학과

1996. 12. 4

## 1 서론

프로그래밍 언어에서 타입의 역할은 여러 가지가 있다. 실행시 각 변수의 기억장소 할당 문제에서 타입이 주는 정보로 효율적인 할당을 할 수 있고, 타입을 통한 자료의 추상화로 프로그램 표현이 자연스럽고, 모듈화된 프로그램 작성시 타입 정보가 모듈의 입출력을 간단하게 표현하므로 커다란 프로그램 개발에 중요한 역할을 하고 있다.

그러한 타입의 역할들 중에서 여기서는 타입을 통해 프로그램의 안전을 보장하는 이야기를 해 보고자 한다. 이야기 순서는, 먼저 여기서 말하는 프로그램의 안전이란 무엇인지를 말하고, 타입이 어떻게 프로그램의 안전을 보장한다는 것인지 개괄적인 이야기를 한 후, 안전을 보장하는 예제 언어를 설계해 봄으로써 그 의미를 명확하게 이해해 보고, 마지막으로 타입유추 알고리즘을 통해 프로그램의 안전이 자동으로 검증되는 것에 대해 이야기해 보기로 한다.

## 2 안전성

프로그램이 안전하다는 것을, 그 프로그램이 실행될 때 잘못 다루어지는 값(run-time type error)이 발생하지 않는다는 것으로 정의하자.[1] 즉, 논리형 타입(boolean type) 변수가 있고, 참을 1로 거짓을 0으로 표현한다고 하자. 프로그램이 안전하다면 이 변수는 실행 중에 0 혹은 1 이외의 다른 값은 절대로 가지지 않음을 의미한다.

프로그래밍 언어가 안전하다는 것은, 그 언어로 짜여진 모든 프로그램의 안전을 미리 검증할 수 있는 것을 뜻한다. 안전하지 않은 언어와 안전한 언어의 예를 들어보자.

C 언어는 안전하지 않다. C에서 다음과 같은 타입을 정의한다고 하자:

```
typedef enum { RED, BLUE } color;
```

C 언어는 새로운 타입을 생성하는 것을 허용하지 않으므로 `color` 타입은 단순히 `RED`는 `0`, `BLUE`는 `1`인 정수를 나타내는 데 불과하다. 따라서, 다음의 코드는 가능하다:

```
int colorInt(color s) {
   switch(s) {
      case RED :   return 1;
      case BLUE :  return 2;
   }
};

colorInt(3);
```

*이 글은 정보과학회지 특집 "차세대 프로그래밍 언어"에 실기 위해 작성된 것입니다.

†{cookcu,nature,kwang}@pllab.kaist.ac.kr, http://compiler.kaist.ac.kr/~{cookcu,nature,kwang}

[1] "안전"을 다른 뜻으로 사용할 수 있다.[1]

위의 코드가 허용됨은 실행 중에 `color` 타입을 갖는 변수 `s`가 정수 타입에 속하는 `3`이라는 잘못된 값을 가질 수 있음을 보여준다.

반면에 Standard ML[4, 6]과 같이 안전한 언어도 있다. Standard ML에서는 앞의 예와 같은 타입을 다음과 같이 정의할 수 있다:

```
datatype color = RED | BLUE
```

Standard ML은 `datatype`에 의해 새로운 타입을 생성하므로 `color` 타입의 변수는 `RED`나 `BLUE`이외의 다른 값은 가질 수 없다. 다음의 함수는 타입이 `color -> int` 이므로, `3`이라는 `int`값에 적용될 수 없으므로 타입 오류가 발생한다.

```
(fn x => (case x of
              RED => 1
            | BLUE => 2 )) 3
```

즉, 타입에 맞지 않게 사용되는 경우를 가진 프로그램은 컴파일 시간에 잡아 냄으로써 실행될 수가 없다.

# 3 안전 보장의 메커니즘

새로운 타입 언어를 설계하려 할 때, 어떻게 하면 타입을 통해 안전을 보장하게 할 수 있고 또한 안전을 보장한다는 것을 어떻게 증명할 수 있을까?

## 3.1 타입을 통한 안전 보장

이에 대한 개략적인 방법을 이야기를 풀어 나가는 순서대로 보면 다음과 같다. 언어의 문법을 정의하고 이 언어로 짜여진 프로그램의 실행이 어떻게 이루어지는지 기술한다. 이러한 언어로 짜여진 프로그램의 실행을 기술한 것을 동적 의미구조(dynamic semantic)라 한다. 그리고, 이 언어의 타입시스템을 정의한다. 이를 대개 정적 의미구조(static semantic)라 한다. 정적 의미구조를 정의할 때, 정적 의미구조를 제대로 따르는 모든 프로그램이 동적의미구조를 제대로 따라 안전하게 실행되도록 한다. 그러면, 이러한 정적 의미구조를 가진 언어는 안전하게 된다. 그리고, 컴파일러 구현시에 정적 의미구조를 충실히 구현하는 타입유추(type inference) 알고리즘을 넣어 프로그램을 검증함으로써 타입유추를 통과한 모든 프로그램에 대해 안전을 보장할 수 있는 것이다.

## 3.2 예제 언어로 보는 안전 보장

이런 메커니즘을 명확하게 이해하기 위해 간단한 언어로 전체 과정을 설계해 보자. 그림 1은 예제로 쓸 간단한 언어이다.

$$
\begin{array}{llcll}
Expr & e & := & \mathtt{1} & ;\ \text{상수} \\
 & & \mid & x & ;\ \text{변수} \\
 & & \mid & \lambda x.e & ;\ \text{함수} \\
 & & \mid & e\ e & ;\ \text{함수적용} \\
Type & \tau & := & \iota & ;\ \text{타입상수} \\
 & & \mid & \tau \rightarrow \tau & ;\ \text{함수타입} \\
Value & v & := & \underline{\mathtt{1}} & ;\ \text{상수값} \\
 & & \mid & \underline{\lambda x.e} & ;\ \text{함수값}
\end{array}
$$

그림 1: 예제 언어

그림 2는 이 언어로 짜여진 프로그램이 어떻게 실행되는지를 기술한 동적의미구조이다.

$$1 \Rightarrow \underline{1}$$

$$\lambda x.e \Rightarrow \underline{\lambda x.e}$$

$$\frac{e_1 \Rightarrow \underline{\lambda x.e},\ e_2 \Rightarrow \underline{v}, [\underline{v}/x]e \Rightarrow \underline{v'}}{e_1\ e_2 \Rightarrow \underline{v'}}$$

그림 2: 예제 언어의 동적의미구조

$$\begin{array}{lcl}
\texttt{λx.λy.y 1 λx.1} & \Rightarrow & \underline{\texttt{λx.λy.y}}\ \texttt{1 λx.1} \\
 & \Rightarrow & \underline{\texttt{λx.λy.y}}\ \underline{\texttt{1}}\ \texttt{λx.1} \\
 & \Rightarrow & \overline{\texttt{λy.y λx.1}} \\
 & \Rightarrow & \underline{\texttt{λy.y}}\ \texttt{λx.1} \\
 & \Rightarrow & \underline{\texttt{λy.y}}\ \underline{\texttt{λx.1}} \\
 & \Rightarrow & \underline{\texttt{λx.1}}
\end{array}$$

그림 3: 동적 의미구조의 실행 예

동적의미구조에 나오는 수식들의 의미를 알아보자. $\Rightarrow$는 프로그램의 실행(evaluation)을 의미한다. 어떤 프로그램 $p$가 동적의미구조에 따라 $p \Rightarrow \ldots \Rightarrow v$가 된다면, $p$가 실행되어 $v$의 결과를 준다는 뜻이다. 세 번째의 식, 분수 형태로 쓰여진 식은 위쪽의 조건이 만족하면 아래쪽처럼 실행된다는 뜻이고, $[\underline{v}/x]e$는 $e$ 내의 모든 $x$를 $\underline{v}$로 바꾸어 준 것이다. 예를 들어 프로그램 **λx.λy.y 1 λx.1**은 위의 동적의미구조에 따라 실행하면 그림 3과 같은 결과를 얻는다. 여기서 실행의 순서는 크게 의미가 없다.

이제 이 언어의 정적 의미구조를 정의해 보자. 정적 의미구조에서 정의하려고 하는 것은 어떤 프로그램의 모든 부분들이 일관되게 타입을 가지고 있는가이다. 언어에서 1은 상수타입이고 $\iota$로 나타낸다. $\lambda x.e$는 함수타입이어야 하므로 어떤 타입 $\tau_1 \to \tau_2$이어야 한다. $e_1\ e_2$의 경우는 복잡한 제약이 생긴다. 우선 $e_1$은 함수 타입이어야 하고 그 함수 타입은 $e_2$의 타입을 받아서 다른 어떤 타입을 주는 타입이어야 한다. 이것을 정확하게 다음과 같이 기술한다:

$$\frac{e_1 : \tau_1 \to \tau_2,\ e_2 : \tau_1}{e_1\ e_2 : \tau_2}$$

즉, $e_1$이 $\tau_1 \to \tau_2$ 타입이고, $e_2$가 $\tau_1$ 타입이면, $e_1\ e_2$는 $\tau_2$ 타입을 가진다는 뜻이다.

$\lambda x.e$의 경우에는 가정이 필요하게 된다. $\lambda x.e$는 $x$의 타입을 받고 $e$의 타입을 주는 함수타입이어야 한다. 만약 $e$ 내에 $x$가 쓰였다고 해보자. 이러한 $x$에 대한 조건을 기술해 주지 않으면 $e$ 내의 $x$들이 일관된 타입을 가질 수 없게 된다. 그러므로, $e$ 내의 모든 부분에 대한 타입 조건에는 모두 $x$가 어떤 하나의 타입을 가져야 한다는 가정이 있어야 한다.

이러한 가정을 타입환경(type environment)이라 하고 프로그램 변수의 타입을 결정하는 테이블이라고 생각하면 된다. "타입환경 $\Gamma$ 아래에 프로그램 $e$가 타입 $\tau$를 갖는다"를 "$\Gamma \vdash e : \tau$"와 같이 쓴다. "$\Gamma + x : \tau$"라고 표기한 것은 타입환경 $\Gamma$의 가정에 $x$의 타입이 $\tau$라는 가정을 더한 것을 말한다. 만약, $\Gamma$에 $x$에 대한 가정이 있으면 기존의 가정은 무시되고 새로 덧붙여져 "$x : \tau$"라는 가정이 사용된다.

그러면 예제 언어의 정적 의미구조를 다음과 같이 정의할 수 있다.

### 3.3 타입 안전성

정의한 예제 언어가 안전성을 가지는가? 앞서 말했듯이 안전성이란 프로그램이 실행될 때 잘못된 값을 가지지 않는 것이라고 했다. 프로그램의 타입이 그 프로그램이 실행될 때 가질 수 있는 값들의 타입과 같다면, 그 프로그램은 실행될 때 잘못된 값을 가지지 않게 된다. 그러므로 그

$$
\begin{array}{ll}
\text{(CON)} & \Gamma \vdash \mathtt{1} : \iota \\[4pt]
\text{(VAR)} & \dfrac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \\[8pt]
\text{(FN)} & \dfrac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2} \\[8pt]
\text{(APP)} & \dfrac{\Gamma \vdash e_1 : \tau_1 \to \tau_2,\ \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}
\end{array}
$$

그림 4: 예제 언어의 정적 의미구조

프로그램은 안전하다고 말할 수 있다. 정리 1이 그것을 수식으로 표현한 것인데, 이러한 성질을 타입 안전성(type soundness)[7]이라 한다.

**정리 1 (타입 안전성)** *$\emptyset \vdash e : \tau$ 이면, $e \Rightarrow^* v$가 존재하고, $v : \tau$이다.*

즉, 프로그램의 타입이 $\tau$이면 ($\emptyset \vdash e : \tau$) 프로그램이 수행될 수 있고 ($e \Rightarrow^* v$) 그 값의 타입은 $\tau$라는 것이다. 예제 언어에서 $v$의 타입은 $\underline{1} : \iota$, $\emptyset \vdash \lambda x.e : \tau$이면, $\underline{\lambda x.e} : \tau$로 정의한다.v“

예제 언어는 타입 안전성을 만족하므로 안전을 보장한다. 만약 타입 안전성을 만족하지 않는다면 그것은 정적 의미구조가 안전을 보장하기에 불충분하게 정의되었다는 것이다. 그러므로, 타입시스템을 설계하려 할 때, 정적 의미구조가 타입 안전성을 만족하도록 정의해야 하는 것이다.

# 4 타입유추 알고리즘

정적 의미구조를 구현하기 위해서 타입유추(type inference) 알고리즘이 필요하다. 앞의 정적 의미구조를 보면 알 수 있듯이 단지 수식일 뿐 구현하기에 적절하지 않다. 그러므로, 구현하기 위하여 단계적으로 타입유추를 할 수 있는 알고리즘이 필요하다. 타입유추 알고리즘은 앞의 예제 언어와 같은 경우 정적 의미구조로부터 쉽게 나올 수 있지만 조금더 복잡한 경우에는 그렇지 않다. 여기서 복잡한 경우라는 것은 예제 언어와는 달리 하나의 타입으로 여러 개의 타입을 나타낼 수 있는(polymorphic type) 경우를 말한다.

예제 언어의 타입유추 알고리즘을 살펴 보자. 그림 5는 예제 언어의 타입유추 알고리즘[2]이다. 여기서 결과값의 하나인 $Type$은 그림 1의 타입에 타입변수가 추가된 것이고, 알고리즘에 사용된 $S$로 표기된 것은 타입 치환함수(substitution), $\mathcal{U}$로 표기된 것은 두 타입을 같게 하는(unifiction) 알고리즘[5]이다.

타입 치환함수(substitution)는 타입변수에서 타입으로 가는 함수이다. 예를 들어 타입 치환함수 $S = \{\alpha \mapsto \iota\}$, 타입 $\tau = \alpha \to \beta$에 대해, $S\tau$는 $\iota \to \beta$가 되고, 타입환경 $\Gamma = \{x \mapsto \alpha,\ y \mapsto (\alpha \to \beta)\}$에 대해, $S\Gamma$는 $\{x \mapsto \iota,\ y \mapsto (\iota \to \beta)\}$가 된다.

타입을 같게 하는 알고리즘 $\mathcal{U}$는 두 개의 타입을 받아서 두 타입을 같게 할 수 있는 타입 치환함수(unifier)를 주는 알고리즘이다. 즉, $S = \mathcal{U}(\tau_1, \tau_2)$이면 $S\tau_1 = S\tau_2$이다. 만일, $\mathcal{U}$ 알고리즘이 실패하면 타입 오류(type error)가 일어나는데, 주어진 입력에 대해 일관된 타입을 유추할 수 없다는 것이다.

앞서 말한 대로 타입유추 알고리즘은 정적 의미구조를 충실히 구현하는지 증명되어야 한다. 타입유추 알고리즘이 정적 의미구조를 충실히 구현한다는 것은 안전성(soundness)과 완전성(completeness)을 만족한다는 것을 의미한다. [3, 2] 타입유추 알고리즘의 안전성이란 타입유추 알고리즘을 통해 얻어진 결과 타입이 정적 의미구조를 만족하는 성질을 말하고, 완전성이란 정적 의미구조를 만족하는 타입을 타입유추 알고리즘을 통해서도 얻어질 수 있는 성질을 말한다. 이 두 가지 성질을 만족하는 타입유추 알고리즘을 사용하면 정적 의미구조에 따라 타입을 유추한다는 것을 보장할 수 있는 것이다.

정리 2와 정리 3은 $\mathcal{I}$ 알고리즘의 안전성과 완전성을 수식으로 나타낸 것이다.

**정리 2 ($\mathcal{I}$ 알고리즘의 안전성)** *$\mathcal{I}(\Gamma, e) = (S, \tau)$이면 $S\Gamma \vdash e : \tau$이다.*

[2] 이 알고리즘은 $\mathcal{W}$ 알고리즘[3]을 예제 언어에 맞게 간단하게 변형한 것이다.

$$\mathcal{I} : TypEnv \times Expr \rightarrow Subst \times Type$$

$$\begin{array}{lll}
\mathcal{I}(\Gamma, \mathtt{1}) = & (id, \iota) & \\
\mathcal{I}(\Gamma, x) = & (id, \Gamma(x)) & \\
\mathcal{I}(\Gamma, \lambda x.e) = & \text{let} & (S_1, \tau_1) = \mathcal{I}(\Gamma + x : \beta, e), \ \text{new } \beta \\
 & \text{in} & (S_1, S_1\beta \rightarrow \tau_1) \\
\mathcal{I}(\Gamma, e_1 \ e_2) = & \text{let} & (S_1, \tau_1) = \mathcal{I}(\Gamma, e_1) \\
 & & (S_2, \tau_2) = \mathcal{I}(S_1\Gamma, e_2) \\
 & & S_3 = \mathcal{U}(S_2\tau_1, \tau_2 \rightarrow \beta), \ \text{new } \beta \\
 & \text{in} & (S_3S_2S_1, S_3\beta)
\end{array}$$

그림 5: 예제 언어의 타입유추 알고리즘

즉, 프로그램이 주어진 타입환경 $\Gamma$ 아래에 $\mathcal{I}$ 알고리즘으로 타입유추를 하여 얻어진 결과가 $(S, \tau)$이면, $(\mathcal{I}(\Gamma, e) = (S, \tau))$ 타입환경 $S\Gamma$ 아래에 프로그램은 $\tau$의 타입을 갖는다는 $(S\Gamma \vdash e : \tau)$ 것이다. $\Gamma$를 공집합으로 생각하면 이해를 쉽게 할 수 있다. 즉, 프로그램은 $\mathcal{I}$ 알고리즘으로 타입유추하여 얻어진 타입을 갖는다는 것이 된다.

**정리 3 ($\mathcal{I}$ 알고리즘의 완전성)** *$\Gamma' \vdash e : \tau'$이고, $\Gamma' = P\Gamma$를 만족하는 타입 치환함수 $P$가 존재하면, $\mathcal{I}(\Gamma, e) = (S, \tau)$이며, $\Gamma' = RS\Gamma$ 이고 $\tau' = R\tau$인 타입 치환함수 $R$이 존재한다.*

즉, 프로그램이 주어진 타입환경 $\Gamma'$ 아래에 $\tau'$의 타입을 갖고, $(\Gamma \vdash e : \tau')$ $\Gamma' = P\Gamma$를 만족하는 타입 치환함수 $P$가 존재하면, 프로그램을 타입환경 $\Gamma$ 아래에 $\mathcal{I}$ 알고리즘으로 타입유추를 하면 $(S, \tau)$의 결과로 성공하며, $(\mathcal{I}(\Gamma, e) = (S, \tau))$ $\Gamma' = RS\Gamma$ 이고 $\tau' = R\tau$인 타입 치환함수 $R$이 존재한다. 이것도 역시 $\Gamma$를 공집합으로 생각하면 이해를 쉽게 할 수 있다. 즉, 프로그램이 $\tau'$의 타입을 가지면, $\mathcal{I}$ 알고리즘으로 타입유추가 성공하여 타입 $\tau$를 얻을 수 있고, 어떤 타입 치환함수 $R$에 대해 $\tau' = R\tau$를 만족한다는 것이 된다.

위의 안정성과 완전성이 증명될 수 있으므로, $\mathcal{I}$ 알고리즘은 정의된 정적 의미구조를 충실히 구현한 것이다. 즉, $\mathcal{I}$ 알고리즘으로 타입유추에 성공한 프로그램은 안전한 프로그램이다.

## 5 결론

안전한 프로그램 작성을 필요로 할 때, 언어자체가 안전을 보장하지 않는다면, 프로그램 작성자는 자신이 작성한 프로그램을 안전하게 하기 위하여, 많은 생각을 해야하고, 작성 후에도 과연 작성된 프로그램이 안전한지, 불안감을 떨쳐 버릴 수가 없다.

이 글에서는 프로그래밍 언어의 안전을, 그 언어의 타입시스템을 통하여 보장해 줄 수 있음을 간단한 예제 언어를 통하여 이야기 해 보았다.

## 참고 서적


[1] Luca Cardelli. Type systems. `http://www.research.digital.com/SRC/personal/Luca_Cardelli/Papers/TypeSystems.A4.ps`, 1996.

[2] Luis Damas and Robin Milner. Principal type-scheme for functional programs. In *Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

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

[4] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Messachusetts, London, England, 1990.

[5] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, January 1965.

[6] Jeffrey D. Ullman. *Elements of ML Programming*. Prentice Hall, Englewood Cliffs, New Jersey 07632, U. S. A., 1994.

[7] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical Report TR91-160, Rice University, 1992.