

SNU 4541.310 Programming Language

Let-polymorphic Type System

Prof. Kwangkeun Yi

다형 타입 시스템 Polymorphic Type System

다형 타입 추론

추론 규칙의 안전성

추론 규칙의 구현

타입 분석의 정확도 올리기

단순 타입 시스템(*simple type system*)이 “잘 모르겠다” 고 하는 경우를 줄여보자.

- ▶ 타입 시스템에서는

“잘 모르겠다” = “타입방정식의 해가 없다”

- ▶ 다형 타입 시스템(*polymorphic type system*)(\vdash_p)에서는 “잘 모르겠다” 고 하는 경우가 적다, 단순 타입 시스템(*simple type system*)(\vdash) 보다:

$$\Gamma \vdash e : \tau \Rightarrow \Gamma \vdash_p e : \tau$$

- ▶ \vdash_p 는 \vdash 의 “conservative extension” 이라고 함.

부정확한 단순 타입 시스템

$$\frac{\frac{\vdots}{\{f : \tau \rightarrow \tau'\} \vdash f : \tau \rightarrow \tau'} \quad \frac{\vdots}{\{f : \tau \rightarrow \tau'\} \vdash f : \tau}}{\frac{\{f : \tau \rightarrow \tau'\} \vdash f f : \tau'}{\vdash \lambda f. f f : (\tau \rightarrow \tau') \rightarrow \tau'}} \quad \tau = \tau \rightarrow \tau'$$

$$\frac{\frac{\frac{\vdots}{\{f : \tau \rightarrow \tau\} \vdash f : \tau \rightarrow \tau} \quad \frac{\vdots}{\{f : \tau \rightarrow \tau\} \vdash f : \tau}}{\frac{\{f : \tau \rightarrow \tau\} \vdash f f : \tau \rightarrow \tau}{\vdash \lambda f. f f : (\tau \rightarrow \tau) \rightarrow (\tau \rightarrow \tau)}} \quad \vdots}{\vdash (\lambda f. f f)(\lambda x. x) : \tau \rightarrow \tau} \quad \tau = \tau \rightarrow \tau$$

타입을 일반화(*type generalization*) 시키면?

타입(분석 결과 즉, 요약해석 공간의 원소)에 정교한 것이 포함됨:

$$\forall \alpha. \alpha \rightarrow \iota, \quad \forall \alpha_1, \alpha_2. \alpha_1 \rightarrow \alpha_2, \quad \dots$$

그래서

$$\frac{\frac{\frac{\vdots}{\{f : \forall \alpha. \alpha \rightarrow \alpha\} \vdash f : (\iota \rightarrow \iota) \rightarrow (\iota \rightarrow \iota)}}{\{f : \forall \alpha. \alpha \rightarrow \alpha\} \vdash f f : \iota}}{\vdash \lambda f. f f : (\forall \alpha. \alpha \rightarrow \alpha) \rightarrow \iota}}{\frac{\vdots}{\{f : \forall \alpha. \alpha \rightarrow \alpha\} \vdash f : \iota \rightarrow \iota}}$$

이고

$$\frac{\frac{\vdots}{\vdash \lambda f. f f : (\forall \alpha. \alpha \rightarrow \alpha) \rightarrow (\iota \rightarrow \iota)}}{\vdash (\lambda f. f f)(\lambda x. x) : \iota \rightarrow \iota}}{\frac{\vdots}{\vdash \lambda x. x : \iota \rightarrow \iota}}$$

하지만 함부로 일반화를 이용하면

- ▶ 안전하지 않을 뿐더러
- ▶ 완전(*complete*)한 구현이 불가능(*undecidable*):
 - ▶ 2단(*rank 2*)이상의 다형타입(*polymorphism*)은 피해야
 - ▶ 그러나 “완전한” 면을 포기하면 불가능할 것도 없음:
프로그램분석의 기본
- ▶ 아뭏튼, 1단(*rank 1*) 다형타입(*polymorphism*)까지만 이용
 - ▶ 0단(*rank 0*) = 단순타입(*monomorphic types*)
 - ▶ 1단(*rank 1*) = “ \forall ” 이 제일 바깥(*prenex forma*)인 다형타입

n 단 다형타입 σ^n (*rank- n polymorphism*)

$$\begin{aligned}\tau, \sigma^0 &::= \iota \mid \tau \rightarrow \tau \mid \alpha \\ \sigma^{n+1} &::= \sigma^n \mid \forall \alpha. \sigma^{n+1} \\ &\quad \mid \sigma^n \rightarrow \sigma^{n+1}\end{aligned}$$

예)

$\iota \rightarrow \iota$	0단
$\forall \alpha. (\alpha \rightarrow \alpha)$	1단
$(\forall \alpha. \alpha) \rightarrow \iota$	2단
$((\forall \alpha. \alpha) \rightarrow \iota) \rightarrow \iota$	3단

함부로 일반화하면 불안전

$$\frac{\frac{\frac{\vdots}{\{f : \forall \alpha. \alpha \rightarrow \iota\} \vdash f : \iota \rightarrow \iota} \dots}{\{f : \forall \alpha. \alpha \rightarrow \iota\} \vdash f \text{ 1} : \iota} \dots}{\{f : \forall \alpha. \alpha \rightarrow \iota\} \vdash (f \text{ 1}, f \text{ true}) : \iota \times \iota} \quad \vdots}{\vdash \lambda f. (f \text{ 1}, f \text{ true}) : (\forall \alpha. \alpha \rightarrow \iota) \rightarrow (\iota \times \iota)} \quad \vdash \lambda x. x + 1 : \iota \rightarrow \iota}{\vdash (\lambda f. (f \text{ 1}, f \text{ true}))(\lambda x. x + 1) : \iota \times \iota}$$

함
부
로

$$\frac{\vdots}{\vdash (\lambda x. (\text{let } y = x \text{ in } (y \text{ 1}, y \text{ true}))) (\lambda z. z + 1) : \iota \times \text{bool}}$$

다형 타입의 의미 $\gamma(\forall\alpha.\tau)$?

값들의 집합

- ▶ 모든 타입 α 에 대해서 τ 타입인 값들
- ▶ 예를들어, $\forall\alpha.\alpha \rightarrow \alpha$ 의 의미는: 인자타입에 상관없이 일을 하고 인자타입과 같은 타입의 값을 리턴하는 함수들의 집합
- ▶ 즉,

$$\gamma(\forall\alpha.\tau) = \bigcap_{t \in SimpleType} \gamma(\{t/\alpha\}\tau)$$

예를들어,

$$\begin{aligned} \gamma(\forall\alpha.\alpha \rightarrow \alpha) &= \bigcap_{t \in SimpleType} \gamma(t \rightarrow t) \\ &= \gamma(\iota \rightarrow \iota) \cap \gamma(bool \rightarrow bool) \cap \dots \\ &= \{\lambda x.x, \lambda x.1, \lambda x.x + 1, \dots\} \cap \\ &\quad \{\lambda x.x, \lambda x.(x || true), \dots\} \cap \dots \end{aligned}$$

안전한 분석 디자인: let-polymorphism 예

“Hindley-Milner style let-polymorphism”

- ▶ 프로그램이 특별히 생긴 경우만 그렇게 정교한 분석이 작동하도록
- ▶ 함수가 어디서 무슨 인자로 어떻게 사용되는 지를 알 수 있는 경우 즉,

$$(\lambda x. \underbrace{\dots x \dots x \dots}_e) e'$$

즉,

$$\text{let } x = e' \text{ in } e$$

인 경우만

- ▶ 이 경우, e' 이 다형타입일 수 있는 지 “안전하게” 분석한 후에, e 안에서 x 가 어떻게 사용되는 지 분석.
- ▶ 다형타입은 1단(rank-1 polymorphism)까지만:

$$\iota \rightarrow \iota, \forall \alpha. \alpha \rightarrow \alpha, \forall \alpha_1, \alpha_2. \alpha_1 \rightarrow \alpha_2$$

다형 타입 추론

- ▶ 타입(*type*)과 타입틀(*type scheme*)

$$\begin{array}{l} \text{Type} \quad \tau \rightarrow \iota \mid \tau \rightarrow \tau \mid \alpha \\ \text{TypeScheme} \quad \sigma \rightarrow \tau \mid \forall \alpha. \sigma \end{array}$$

타입틀(*type scheme*)은 단순타입과 다형타입을 포함.
다형타입은 1단(*rank-1*)까지만 (prenex form).

- ▶ 추론규칙(*inference rules*)은 “ $\Gamma \vdash e : \tau$ ” 꼴을 유추하는 규칙들
- ▶ 가정들 Γ
 - ▶ 변수들의 타입틀(*type scheme*)에 대한 가정
 - ▶ $x + 1 : \iota$, 가정 $x : \iota$ 아래서.
 - ▶ $(f \ 1, f \ \text{true}) : \iota \times \text{bool}$, 가정 $f : \forall \alpha. \alpha \rightarrow \alpha$ 아래서

$\Gamma \vdash e : \tau$ 를 추론하는 규칙

$$\overline{\Gamma \vdash n : \iota} \quad \overline{\Gamma \vdash x : \tau} \quad \sigma \succ \tau, x : \sigma \in \Gamma$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma + x : \text{Gen}_{\Gamma}(\tau) \vdash e' : \tau'}{\Gamma \vdash \text{let } x = e \text{ in } e' : \tau'}$$

$$\frac{\Gamma \vdash e_1 : \iota \quad \Gamma \vdash e_2 : \iota}{\Gamma \vdash e_1 + e_2 : \iota}$$

$$\frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 e_2 : \tau}$$

$$\frac{\Gamma + x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x. e : \tau \rightarrow \tau'}$$

generalization $\text{Gen}_{\Gamma}(\tau) = \forall \alpha_1, \dots, \alpha_n. \tau$ $\{\alpha_1, \dots, \alpha_n\} = \text{FTV}(\tau) \setminus \text{FTV}(\Gamma)$
instantiation $\sigma \succ \tau$ $\sigma = \forall \alpha_1, \dots, \alpha_n. \tau', \tau = \{\tau_i / \alpha_i\}_i \tau'$

$$\text{FTV}(\tau) = \text{TV}(\tau)$$

$$\text{FTV}(\forall \alpha. \sigma) = \text{FTV}\sigma \setminus \{\alpha\}$$

$$\text{FTV}(\Gamma) = \cup_{x:\sigma \in \Gamma} \text{FTV}(\sigma)$$

추론 규칙의 안전성: $Gen_{\Gamma}(\tau)$

왜 타입 τ 를 일반화 $\forall\alpha.\tau$ 시키는데 α 가 Γ 에 나타나면 제외?

$$Gen_{\Gamma}(\tau) = \forall\alpha_1, \dots, \alpha_n.\tau \quad \text{여기서 } \{\alpha_1, \dots, \alpha_n\} = FTV(\tau) \setminus FTV(\Gamma)$$

- ▶ Γ 에 가정($x : \sigma$)이 첨가 되는 경우는 $\lambda x.e$ 의 경우
- ▶ Γ 에 있는 가정을 사용하는 경우는 함수안에서 함수의 인자를 분석할 때
- ▶ 함수의 인자 타입을 일반화시키고나서 함수 내부가 분석되면 안되
- ▶ 함수가 호출되면서 전달받는 실제인자는 일반화된 타입의 값이 아닐 수 있기 때문

예를 들어,

$$\lambda x.(\text{let } y = x \text{ in } (y \ 1, y \ \text{true}))$$

이고 이 함수가 $\lambda z.z + 1$ 에 적용되면?

추론 규칙의 안전성 증명: 추론되는 대로 실행된다

타입이 있으면 문제없이 진행:

Theorem (Progress)

$\vdash e : \tau$ 이고 e 가 값이 아니면 반드시 진행 $e \rightarrow e'$ 한다.

진행은 타입을 보존:

Theorem (Preservation)

$\vdash e : \tau$ 이고 $e \rightarrow e'$ 이면 $\vdash e' : \tau$.

추론 규칙의 구현

- ▶ 단순타입 유추 알고리즘 M 과 W 의 “자연스러운” 확장
- ▶ M_p 나 W_p 도 모두 충실한 구현. 예를들어,

$$\begin{array}{l} \boxed{\text{안전(sound)}} \\ \boxed{\text{완전(complete)}} \end{array} \left. \begin{array}{l} W_p(\Gamma, e) = (\tau, S) \Rightarrow S\Gamma \vdash e : \tau \\ W_p(\Gamma, e) = (\tau, S) \\ \wedge \Gamma' = RST \\ \wedge R(\text{Gen}_{S\Gamma}(\tau)) \succ \tau' \end{array} \right\} \Leftarrow \Gamma' \vdash e : \tau'$$

참고: “Proofs about a Folklore Let-Polymorphic Type Inference Algorithm”, Oukseh Lee and Kwangkeun Yi, *TOPLAS*, 20(4), 1998

let-다형 타입 유추 알고리즘 \mathcal{W}

$$\mathcal{W} : TyEnv \times Exp \rightarrow ((TyVar \xrightarrow{\text{fin}} Type) \times Type)$$

$$\mathcal{W}(\Gamma, n) = (\emptyset, \iota)$$

$$\mathcal{W}(\Gamma, x) = (\emptyset, \{\alpha_i \mapsto \beta_i\}_{i=1}^n \tau) \quad \text{where } \Gamma(x) = \forall \vec{\alpha}. \tau, \text{ new } \vec{\beta}$$

$$\begin{aligned} \mathcal{W}(\Gamma, \lambda x. E) &= \text{let } (S_1, \tau_1) = \mathcal{W}(\Gamma + x : \beta, E), \text{ new } \beta \\ &\text{in } (S_1, S_1 \beta \rightarrow \tau_1) \end{aligned}$$

$$\begin{aligned} \mathcal{W}(\Gamma, E_1 E_2) &= \text{let } (S_1, \tau_1) = \mathcal{W}(\Gamma, E_1) \\ &\quad (S_2, \tau_2) = \mathcal{W}(S_1 \Gamma, E_2) \\ &\quad S_3 = \mathcal{U}(S_2 \tau_1, \tau_2 \rightarrow \beta), \text{ new } \beta \\ &\text{in } (S_3 S_2 S_1, S_3 \beta) \end{aligned}$$

$$\begin{aligned} \mathcal{W}(\Gamma, \text{let } x = E_1 \text{ in } E_2) &= \\ &\text{let } (S_1, \tau_1) = \mathcal{W}(\Gamma, E_1) \\ &\quad (S_2, \tau_2) = \mathcal{W}(S_1 \Gamma + x : GEN_{S_1 \Gamma}(\tau_1), E_2) \\ &\text{in } (S_2 S_1, \tau_2) \end{aligned}$$

let-다형 타입 유추 알고리즘 \mathcal{M}

$$\mathcal{M} : TyEnv \times Exp \times Type \rightarrow (TyVar \xrightarrow{\text{fin}} Type)$$

$$\mathcal{M}(\Gamma, n, \tau) = \mathcal{U}(\tau, \iota)$$

$$\mathcal{M}(\Gamma, x, \tau) = \mathcal{U}(\tau, \{\alpha_i \mapsto \beta_i\}_{i=1}^n \tau') \quad \text{where } \Gamma(x) = \forall \vec{\alpha}. \tau', \text{ new } \vec{\beta}$$

$$\begin{aligned} \mathcal{M}(\Gamma, \lambda x. E, \tau) &= \text{let } S_1 = \mathcal{U}(\tau, \beta_1 \rightarrow \beta_2), \text{ new } \beta_1, \beta_2 \\ &\quad S_2 = \mathcal{M}(S_1\Gamma + x : S_1\beta_1, E, S_1\beta_2) \\ &\quad \text{in } S_2S_1 \end{aligned}$$

$$\begin{aligned} \mathcal{M}(\Gamma, E_1 E_2, \tau) &= \text{let } S_1 = \mathcal{M}(\Gamma, E_1, \beta \rightarrow \tau), \text{ new } \beta \\ &\quad S_2 = \mathcal{M}(S_1\Gamma, E_2, S_1\beta) \\ &\quad \text{in } S_2S_1 \end{aligned}$$

$$\begin{aligned} \mathcal{M}(\Gamma, \text{let } x = E_1 \text{ in } E_2, \tau) &= \\ &\quad \text{let } S_1 = \mathcal{M}(\Gamma, E_1, \beta), \text{ new } \beta \\ &\quad S_2 = \mathcal{M}(S_1\Gamma + x : GEN_{S_1\Gamma}(S_1\beta), E_2, S_1\tau) \\ &\quad \text{in } S_2S_1 \end{aligned}$$

Polymorphic Imperatives

$$E \rightarrow \begin{array}{l} \vdots \\ | \text{ malloc } E \\ | !E \\ | E := E \end{array}$$
$$\text{Type } \tau \rightarrow \begin{array}{ll} \iota & \text{primitive type} \\ | \alpha & \text{type variable} \\ | \tau \rightarrow \tau & \text{function type} \\ | \tau \text{ loc} & \text{pointer type} \end{array}$$
$$\text{TypeScheme } \sigma \rightarrow \begin{array}{ll} \tau & \text{simple type} \\ | \forall \alpha. \sigma & \text{generalized type} \end{array}$$

Let-polymorphic Imperatives: trial

$$\frac{\Gamma \vdash E : \tau}{\Gamma \vdash \text{malloc } E : \tau \text{ loc}}$$

$$\frac{\Gamma \vdash E : \tau \text{ loc}}{\Gamma \vdash !E : \tau}$$

$$\frac{\Gamma \vdash E_1 : \tau \text{ loc} \quad \Gamma \vdash E_2 : \tau}{\Gamma \vdash E_1 := E_2 : \tau}$$

$$\frac{\Gamma \vdash E_1 : \tau_1 \quad \Gamma \vdash E_2 : \tau_2}{\Gamma \vdash E_1 ; E_2 : \tau_2}$$

이번에도 일반화 할 때 조심스러워야:

$$\frac{\Gamma \vdash E : \tau \quad \Gamma + x : \text{GEN}_{\Gamma}(\tau) \vdash E' : \tau'}{\Gamma \vdash \text{let } x = E \text{ in } E' : \tau'}$$

식 E 가 실행중에 메모리 주소를 새롭게 할당받는 일이 없는 경우에만 안전하다.

Unsound typing example

```
let  
  f = malloc ( $\lambda x.x$ )  
in  
  f :=  $\lambda x.x+1$ ;  
  (!f) true
```

- ▶ 실행중 타입 에러.
- ▶ 그러나 우리의 타입 시스템은 타입이 있는 것으로 허용.

- ▶ 식 E 가 실행중에 메모리 주소를 할당받을 지를 유추해야.
- ▶ 실행전에 정확히는 할 수 없다.
- ▶ 하지만, 안전하게는 판단할 수 있다.

그래서 `let`-식의 타입 유추 규칙이 다음과 같이 세분화 된다:

$$\frac{\Gamma \vdash E : \tau \quad \Gamma + x : GEN_{\Gamma}(\tau) \vdash E' : \tau'}{\Gamma \vdash \text{let } x = E \text{ in } E' : \tau'} \neg \text{expansive}(E)$$

$$\frac{\Gamma \vdash E : \tau \quad \Gamma + x : \tau \vdash E' : \tau'}{\Gamma \vdash \text{let } x = E \text{ in } E' : \tau'} \text{expansive}(E)$$

$$\text{expansive}(n) = \text{false}$$

$$\text{expansive}(x) = \text{false}$$

$$\text{expansive}(\lambda x.E) = \text{false}$$

$$\text{expansive}(E_1 E_2) = \text{true}$$

$$\text{expansive}(\text{let } x = E_1 \text{ in } E_2) = \text{expansive}(E_1) \vee \text{expansive}(E_2)$$

Let-polymorphic Imperatives

$$\frac{\Gamma \vdash E : \tau}{\Gamma \vdash \text{malloc } E : \tau \text{ loc}}$$

$$\frac{\Gamma \vdash E : \tau \text{ loc}}{\Gamma \vdash !E : \tau}$$

$$\frac{\Gamma \vdash E_1 : \tau \text{ loc} \quad \Gamma \vdash E_2 : \tau}{\Gamma \vdash E_1 := E_2 : \tau}$$

$$\frac{\Gamma \vdash E_1 : \tau_1 \quad \Gamma \vdash E_2 : \tau_2}{\Gamma \vdash E_1 ; E_2 : \tau_2}$$

$$\frac{\Gamma \vdash E : \tau \quad \Gamma + x : \text{GEN}_{\Gamma}(\tau) \vdash E' : \tau'}{\Gamma \vdash \text{let } x = E \text{ in } E' : \tau'} \neg \text{expansive}(E)$$

$$\frac{\Gamma \vdash E : \tau \quad \Gamma + x : \tau \vdash E' : \tau'}{\Gamma \vdash \text{let } x = E \text{ in } E' : \tau'} \text{expansive}(E)$$

$$\text{expansive}(n) = \text{false}$$

$$\text{expansive}(x) = \text{false}$$

$$\text{expansive}(\lambda x. E) = \text{false}$$

$$\text{expansive}(E_1 E_2) = \text{true}$$

$$\text{expansive}(\text{let } x = E_1 \text{ in } E_2) = \text{expansive}(E_1) \vee \text{expansive}(E_2)$$