

λ Prolog로 나만의 타입체커 만들기

전남대학교 기계공학부 임기정

2021년 2월 20일

서론

- ▶ 타입체커를 쉽게 만들고자 하시는 분들을 위해 발표합니다!

서론

- ▶ 타입체커를 쉽게 만들고자 하시는 분들을 위해 발표합니다!
- ▶ System-F:
 1. 단순 타입 람다-계산법에 타입에 대한 보편 양화사를 추가한 시스템입니다.
 2. 하스켈과 ML 같은 함수형 언어의 이론적 기반입니다.

서론

- ▶ 타입체커를 쉽게 만들고자 하시는 분들을 위해 발표합니다!
- ▶ System-F:
 1. 단순 타입 람다-계산법에 타입에 대한 보편 양화사를 추가한 시스템입니다.
 2. 하스켈과 ML 같은 함수형 언어의 이론적 기반입니다.
- ▶ 논리형 프로그래밍:
 1. 규칙이라 불리는 논리식들을 작성하는 프로그래밍 패러다임입니다.
 2. 그리하여 인터프리터가 그 규칙들로부터 질의받은 논리식을 증명할 수 있는지를 응답합니다.

서론

- ▶ 타입체커를 쉽게 만들고자 하시는 분들을 위해 발표합니다!
- ▶ System-F:
 1. 단순 타입 람다-계산법에 타입에 대한 보편 양화사를 추가한 시스템입니다.
 2. 하스켈과 ML 같은 함수형 언어의 이론적 기반입니다.
- ▶ 논리형 프로그래밍:
 1. 규칙이라 불리는 논리식들을 작성하는 프로그래밍 패러다임입니다.
 2. 그리하여 인터프리터가 그 규칙들로부터 질의받은 논리식을 증명할 수 있는지를 응답합니다.
- ▶ λ Prolog:
 1. 람다-식을 매칭할 수 있다는 게 가장 두드러지는 특징인 논리형 프로그래밍 언어입니다.
 2. 가장 유명한 구현체로는 Teyjus가 있는데, 본 발표에서는 [\[링크\]](#)에서 다운받을 수 있는 Teyjus V2를 사용할 것입니다.

λ Prolog의 구문론과 의미론

- ▶ λ Prolog의 타입체계는 $\text{System-}F_\omega$ 에 랭크 1의 다형성만을 허용한 것입니다.
- ▶ 변수 x : 하스켈과 반대로, 대문자로 시작하는 식별자입니다.
- ▶ 상수 c : 하스켈과 반대로, 소문자로 시작하는 식별자입니다.
- ▶ 항 t :
 1. $t ::= x$ 변수는 항입니다.
 2. $t ::= c$ 상수는 항입니다.
 3. $t ::= (t\ t)$ 적용은 항입니다.
 4. $t ::= (x \setminus t)$ λ -식은 항입니다.
- ▶ 술어 p : 타입 $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow o$ 의 변수 또는 상수입니다 – 단, $n \geq 0$ 이고 타입 τ_1, \dots, τ_n 에는 타입 생성자 o 가 나타나지 않습니다 – 이때 p 를 n 항 술어라고 합니다.
- ▶ 원자논리식 $A: p\ t_1 \cdots t_n$ 꼴의 항입니다 – 여기서, p 는 n 항 술어입니다.
- ▶ 엄격한 원자논리식 A_r : 술어가 상수인 원자논리식입니다.

λ Prolog의 구문론과 의미론

- ▶ 질의 G :
- ▶ $G ::= A$ 는 규칙에 의하여 증명되는 질의입니다.
- ▶ $G ::= \text{true}$ 은 바로 증명되는 질의입니다.
- ▶ $G ::= G_1, G_2$ 은 G_1 과 G_2 모두를 증명해야 증명되는 질의입니다.
- ▶ $G ::= G_1; G_2$ 은 G_1 과 G_2 중 하나만 증명해도 증명되는 질의입니다.
- ▶ $G ::= D_1 \Rightarrow G_2$ 은 D_1 을 가정했을 때 G_2 을 증명해야 증명되는 질의입니다.
- ▶ $G ::= \text{pi } (x_1 \setminus G_2)$ 은 새로운 상수 기호 c 을 도입하여 $G_2[x_1 := c]$ 을 증명해야 증명되는 질의입니다.
- ▶ $G ::= \text{sigma } (x_1 \setminus G_2)$ 은 새로운 논리 변수 X 를 도입하여 $G_2[x_1 := X]$ 를 증명해야 증명되는 질의입니다.

λ Prolog의 구문론과 의미론

- ▶ 규칙 D :
- ▶ $D ::= \text{pi } (x_1 \setminus D_2)$ 는 머리와 꼬리가 $D_2[x_1 := X]$ 와 같은 규칙입니다 – 단, 여기서 X 는 매칭을 시도할 때마다 새롭게 주어지는 논리 변수입니다.
- ▶ $D ::= A_r$ 은 머리가 A_r 이고 꼬리가 true 인 규칙입니다.
- ▶ $D ::= A_r :- G$ 은 머리가 A_r 이고 꼬리가 G 인 규칙입니다.
- ▶ λ Prolog 인터프리터는 그것이 증명하고자 하는 원자논리식을 각 규칙들의 머리에 매칭하길 시도하고, 매칭에 성공한 규칙들의 꼬리의 증명을 시도합니다.

시그니처

- ▶ 이제 타입체커의 시그니처 파일을 다음과 같이 작성해봅시다.

```
% file-name = "systemf.sig"
sig systemf.

kind ty (type).                % data Ty
type all ((ty -> ty) -> ty). %   = All (Ty -> Ty)
type arr (ty -> ty -> ty).    %   | Arr Ty Ty

kind tm (type).                % data Tm
type iabs (ty -> (tm -> tm) -> tm). %   = IAbs Ty (Tm -> Tm)
type iapp (tm -> tm -> tm).      %   | IApp Tm Tm
type tabs ((ty -> tm) -> tm).    %   | TAbs (Ty -> Tm)
type tapp (tm -> ty -> tm).      %   | TApp Tm Ty

type copy_ty (ty -> ty -> o). % copy_ty Ty1 Ty2 <=> Ty1 == Ty2
type check (tm -> ty -> o).  % check Tm Ty <=> |- Tm : Ty

end
```

시그니처

- ▶ 이제부터 이 코드의 의미를 알아보시다.
- ▶ 우리는 닫힌 타입과 닫힌 항만의 System-F의 타입체커를 짤 것입니다 – 즉, 우리가 다룰 항과 타입에 자유변수가 없다고 가정합니다.

시그니처

- ▶ 먼저, System-F의 타입을 인코딩한 타입 생성자 ty 에 대하여 알아보시다.
- ▶ ty 의 kind가 $*$ 로 선언되어 있고, 그 밑으로는 ty 의 자료 생성자가 선언되어 있습니다.
- ▶ $\text{all } (\alpha \setminus \sigma)$ 는 $(\forall \alpha. \sigma)$ 를 인코딩한 것이며,
- ▶ $\text{arr } \tau \ \sigma$ 는 $(\tau \rightarrow \sigma)$ 를 인코딩한 것입니다.

시그니처

- ▶ 그 다음, System-F의 항을 인코딩한 타입 생성자 tm 에 대하여 알아보시다.
- ▶ tm 의 kind가 $*$ 로 선언되어 있고, 그 밑으로는 tm 의 자료 생성자가 선언되어 있습니다.
- ▶ $\mathsf{iabs} \ \tau \ (x \setminus M)$ 은 $(\lambda x^\tau. M)$ 을 인코딩한 것입니다.
- ▶ $\mathsf{iapp} \ M \ N$ 은 (MN) 을 인코딩한 것입니다.
- ▶ $\mathsf{tabs} \ (\alpha \setminus M)$ 은 $(\Lambda \alpha. M)$ 을 인코딩한 것입니다.
- ▶ $\mathsf{tapp} \ M \ \tau$ 는 $(M\tau)$ 를 인코딩한 것입니다.

시그니처

- ▶ 마지막으로, 다음 두 술어가 선언되어 있는 게 보이네요:
- ▶ `copy_ty`: 타입 $ty \rightarrow ty \rightarrow o$ 의 술어로서, $\sigma[\alpha := \tau]$ 와 τ 가 주어졌을 때 가능한 σ 를 모두 찾기 위해 필요합니다.
- ▶ `check`: 타입 $tm \rightarrow ty \rightarrow o$ 의 술어로서, $\vdash M : \sigma$ 일 때 그리고 그럴 때에만 `check M σ` 이 성립하도록 구현할 것입니다.

- ▶ 이제 타입체커의 모듈 파일을 다음과 같이 작성해봅시다.

```
% file-name = "systemf.mod"
module systemf.

copy_ty (arr Tau1 Sigma1) (arr Tau2 Sigma2) :- copy_ty Tau1 Tau2, c
opy_ty Sigma1 Sigma2.
copy_ty (all Sigma1) (all Sigma2) :- pi (Alpha\ copy_ty Alpha Alpha
=> copy_ty (Sigma1 Alpha) (Sigma2 Alpha)).

check (iabs Tau Tm) (arr Tau Sigma) :- pi (Var\ check Var Tau => ch
eck (Tm Var) Sigma).
check (iapp Tm1 Tm2) Sigma :- check Tm1 (arr Tau Sigma), check Tm2
Tau.
check (tabs Tm) (all Sigma) :- pi (Alpha\ copy_ty Alpha Alpha => ch
eck (Tm Alpha) (Sigma Alpha)).
check (tapp Tm Tau) Sigma1 :- pi (Alpha\ copy_ty Alpha Tau => copy_
ty (Sigma Alpha) Sigma1, check Tm (all Sigma)).

end
```

구현

- ▶ 먼저, `copy_ty`의 구현부터 봅시다.

구현

- ▶ 먼저, `copy_ty`의 구현부터 봅시다.
- ▶ 첫 번째 규칙의 의미는 그리 어렵지 않네요:
 1. τ_1 이 τ_2 의 복사본이고
 2. σ_1 이 σ_2 의 복사본이면,
 3. $\tau_1 \rightarrow \sigma_1$ 은 $\tau_2 \rightarrow \sigma_2$ 의 복사본이다.

구현

- ▶ 먼저, `copy_ty`의 구현부터 봅시다.
- ▶ 첫 번째 규칙의 의미는 그리 어렵지 않네요:
 1. τ_1 이 τ_2 의 복사본이고
 2. σ_1 이 σ_2 의 복사본이면,
 3. $\tau_1 \rightarrow \sigma_1$ 은 $\tau_2 \rightarrow \sigma_2$ 의 복사본이다.
- ▶ 두 번째 규칙의 의미를 살펴봅시다:
 1. 타입 `ty`의 새로운 상수 기호 c 를 도입하고, c 는 c 의 복사본이라고 하자.
 2. 이때, $\tau_1 [\alpha := c]$ 가 $\tau_2 [\alpha := c]$ 의 복사본이면,
 3. $\forall \alpha. \tau_1$ 는 $\forall \alpha. \tau_2$ 의 복사본이다.
- ▶ 이렇게 새로운 상수 기호 c 를 도입하는 이유는 재귀를 하기 위해 `ty` \rightarrow `ty`를 `ty`로 바꿔주기 위함이고,
- ▶ c 는 c 의 복사본이라는 규칙을 추가하는 이유는 재귀의 기저 단계에서 필요하기 때문입니다.

구현

- ▶ 먼저, `copy_ty`의 구현부터 봅시다.
- ▶ 첫 번째 규칙의 의미는 그리 어렵지 않네요:
 1. τ_1 이 τ_2 의 복사본이고
 2. σ_1 이 σ_2 의 복사본이면,
 3. $\tau_1 \rightarrow \sigma_1$ 은 $\tau_2 \rightarrow \sigma_2$ 의 복사본이다.
- ▶ 두 번째 규칙의 의미를 살펴봅시다:
 1. 타입 `ty`의 새로운 상수 기호 c 를 도입하고, c 는 c 의 복사본이라고 하자.
 2. 이때, $\tau_1 [\alpha := c]$ 가 $\tau_2 [\alpha := c]$ 의 복사본이면,
 3. $\forall \alpha. \tau_1$ 는 $\forall \alpha. \tau_2$ 의 복사본이다.
- ▶ 이렇게 새로운 상수 기호 c 를 도입하는 이유는 재귀를 하기 위해 `ty` \rightarrow `ty`를 `ty`로 바꿔주기 위함이고,
- ▶ c 는 c 의 복사본이라는 규칙을 추가하는 이유는 재귀의 기저 단계에서 필요하기 때문입니다.
- ▶ 이제 `check`의 구현을 살펴보겠습니다.

구현

- ▶ 첫 번째 줄을 해석하면 다음과 같습니다:
 1. 타입 tm 의 새로운 상수 기호 c 를 도입하고, $c : \tau$ 라고 하자.
 2. 이때, $M[x := c] : \sigma$ 이면,
 3. $\lambda x^\tau. M : \tau \rightarrow \sigma$ 이다.
- ▶ 새로운 상수 기호 c 를 도입하는 이유는 마찬가지로 재귀하기 위해서이고,
- ▶ $c : \tau$ 라고 선언하는 이유는 역시 마찬가지로 재귀의 기저 단계에서 필요하기 때문입니다.

구현

▶ 두 번째 줄을 해석하면 다음과 같습니다:

1. $M : \tau \rightarrow \sigma$ 이고
2. $N : \tau$ 이면,
3. $MN : \sigma$ 이다.

구현

- ▶ 세 번째 줄을 해석하면 다음과 같습니다:
 1. 타입 ty 의 새로운 상수 c 를 도입하고, c 는 c 의 복사본이라고 하자.
 2. 이때, $M[\alpha := c] : \sigma[\alpha := c]$ 이면,
 3. $\Lambda\alpha.M : \forall\alpha.\sigma$ 이다.
- ▶ c 를 도입하는 이유는 역시 앞의 규칙들과 같습니다.
- ▶ c 를 c 의 복사본이라고 하는 이유 또한 앞서와 마찬가지로 copy_ty 의 기저 단계에서 쓰기 위함입니다.

구현

- ▶ 네 번째 줄을 해석하면 다음과 같습니다:
 1. 타입 ty 의 새로운 상수 c 를 도입하고, c 를 τ 의 복사본이라고 하자.
 2. 이때, $\sigma[\alpha := c]$ 이 σ_1 의 복사본이고
 3. $M : \forall \alpha. \sigma$ 이면,
 4. $M\tau : \sigma_1$ 이다.
- ▶ 우리는 tapp에 대한 타이핑 규칙을 “ $M : \forall \alpha. \sigma$ 이면 $M\tau : \sigma[\alpha := \tau]$ 이다”라는 규칙으로 알고 있습니다.
- ▶ 그런데 이 규칙이 저 규칙과 같을까요? 네, 그렇습니다.

구현

- ▶ 먼저, 인터프리터는 c 가 τ 의 복사본이라고 가정했을 때, $\sigma[\alpha := c]$ 가 σ_1 의 복사본이 되게 하는 σ 를 찾으려고 합니다.
- ▶ 이때 σ_1 안에서의 τ 의 나타남이
 1. 기존의 copy_ty 규칙에 의하여 τ 로 복사되어 $\sigma[\alpha := c]$ 의 부분항이 되는 경우도 찾아내고,
 2. c 가 τ 의 복사본이라는 규칙에 의하여 c 로 복사되어 $\sigma[\alpha := c]$ 의 부분항이 되는 경우도 찾아냅니다.
- ▶ σ_1 의 나머지 부분은 copy_ty에 의하여 $\sigma[\alpha := c]$ 로 복사되므로,
- ▶ 인터프리터는 $\sigma_1 \equiv \sigma[\alpha := \tau]$ 인 σ 를 모두 찾아냅니다.
- ▶ 그 다음, 인터프리터는 이렇게 얻은 모든 σ 들에 대하여 $M : \forall \alpha. \sigma$ 인지 조사하기 때문에, 네 번째 규칙은 우리가 알고 있는 tapp에 대한 규칙과 결국 같은 규칙임을 알 수 있습니다.

실행

- ▶ 이제, 실행해 봅시다.
- ▶ 먼저, 컴파일러 'tjcc', 링커 'tjlink', 시뮬레이터 'tjsim'을 앞의 링크에서 다운받습니다.
- ▶ 그리고 다음과 같이 입력하세요:
 1. `tjcc systemf`
 2. `tjlink systemf`
 3. `tjsim systemf`
- ▶ 그러면 인터프리터가 실행됩니다.

실행

```
C:\Users\user\Repository>tjcc systemf
```

```
C:\Users\user\Repository>tjlink systemf
```

```
C:\Users\user\Repository>tjsim systemf
```

```
Welcome to Teyjus
```

```
Copyright (C) 2008 A. Gacek, S. Holte, G. Nadathur, X. Qi, Z. Snow
```

```
Teyjus comes with ABSOLUTELY NO WARRANTY
```

```
This is free software, and you are welcome to redistribute it  
under certain conditions. Please view the accompanying file
```

```
COPYING for more information
```

```
[systemf] ?- check (tapp (tabs (Alpha\ iabs Alpha (X\ X))) (all (Beta\ Beta  
) ) ) (arr (all (Beta\ Beta)) (all (Beta\ Beta))).
```

```
yes
```

```
[systemf] ?- check (tapp (tabs (Alpha\ iabs Alpha (X\ X))) (all (Beta\ Beta  
) ) ) (arr (all (Beta\ Beta)) (arr (all (Beta\ Beta)) (all (Beta\ Beta)))).
```

```
no (more) solutions
```

실행

- ▶ 먼저,

$$\vdash (\Lambda\alpha.\lambda x^\alpha.x) (\forall\beta.\beta) : (\forall\beta.\beta) \rightarrow (\forall\beta.\beta)$$

인지 질의했는데,

- ▶ Teyjus V2는 yes라고 답했네요.
- ▶ 그 다음,

$$\vdash (\Lambda\alpha.\lambda x^\alpha.x) (\forall\beta.\beta) : (\forall\beta.\beta) \rightarrow (\forall\beta.\beta) \rightarrow (\forall\beta.\beta)$$

인지 질의했는데,

- ▶ Teyjus V2는 no (more) solutions라고 답했네요.
- ▶ 두 경우 모두 예상대로 작동하는 것을 알 수 있습니다.

