System F_π

: 패턴을 통한 정제 타입이 추가된 System $F_{<:}$

System F_π: System $F_{<:}$ with Pattern-Based Refined Types

홍재민 류석영
PLRG@KAIST

1 배경

1.1 합 타입 (Sum Type)

- 하나의 타입에 대한 여러 이형(variant)을 정의할 수 있음
- 하나의 타입에 여러 불균일한 형태가 포함되는 경우를 표현

```
type nat = Zero | Succ of nat
let zero: nat = Zero
let one: nat = Succ zero
```

- 패턴 매칭을 통하여 각 형태의 경우로 나누어 타입 에러 없이 처리

```
let pred (n: nat) = match n with
  | Zero -> Zero
  | Succ p -> p
zero == (pred one)   (* true *)
```

1.2 System F

- 간단 타입 람다 대수(simply typed lambda calculus)에 매개변수적 다형성(parametric polymorphism)을 추가하여 확장
- 항을 타입으로 요약 가능

```
let f = λX.λx:X.x in
let n = f [nat] 1 in
f [bool] true
```

1.3 System $F_{<:}$

- System F에 서브타이핑(subtyping)을 추가하여 확장
- 타입 요약 시에 타입 변수의 상한 지정 가능

```
let f = λX<:{a: nat}.λx:X.x in
(f [{a: nat, b: bool}] {a=1, b=true}).b
```

2 동기

- 기존의 pred 함수는 비직관적으로 동작

```
let pred (n: nat) = match n with
  | Zero -> Zero
  | Succ p -> p
zero == (pred zero)   (* true *)
```

- 예외를 사용하여 직관적으로 만들 수 있으나 안전하지 않음

```
exception NoPredForZero
let pred (n: nat) = match n with
  | Zero -> raise NoPredForZero
  | Succ p -> p
pred zero   (* Exception: NoPredForZero *)
```

- pred 함수가 1 이상의 자연수 만을 인자로 받도록 제약
- 합 타입을 이형에 따라 정제(refine)할 수 있도록 언어를 확장

3 System F_π 예시

- 각 이형에 대한 타입을 사용 가능
  - `Succ[Nat]`: `Nat` 타입의 값을 인자로 받은 `Succ` 이형
  - `Succ[Zero[Unit]]`: `Zero[Unit]` 타입의 값을 인자로 받은 `Succ` 이형
- #을 사용해서 패턴을 역적용 가능
  - `(Succ (Zero ()))#Succ[ε]` → `Zero ()`
  - `(Succ (Succ (Zero ())))#Succ[ε]` → `Succ (Zero ())`

```
let Nat = Zero Unit | Succ Nat in
let zero = Zero[Unit] () in
let one = Succ[Zero[Unit]] zero in
let pred = λx:Succ[Nat].x#Succ[ε] in
pred one
```

- pred의 결과 타입은 Nat이기 때문에 부정확

```
let two = Succ[Succ[Zero[Unit]]] one in
pred (pred two)   (* Type Error *)
```

- 매개변수적 다형성을 통해서 정확도 유지 가능

```
let pred = λX<:Nat.λx:Succ[X].x#Succ[ε] in
pred[Nat] (pred[Succ[Nat]] two)   (* Type: Nat *)
```

- 더하기 등의 함수는 정확하게 표현하기 어려움

```
let addNat: Nat -> Nat -> Nat = fix … in
let three = addNat two one in
pred three   (* Type Error *)
```

- 패턴 요약과 패턴 적용을 언어에 추가하여 해결 가능
  - λ_πP.λx:P[Nat].x: 임의의 패턴 P에 대해 사용 가능한 함수
  - `let f = λπP.λx:P[Nat].x in f[Succ]` → `λx:Succ[Nat].x`
  - `Succ[Nat]@zero` → `Succ[Nat] zero`
  - `let f = λπP.P[Nat]@zero in f[Succ]` → `Succ[Nat] zero`

```
let add =
  λπP.λx:P[Nat].λy:Nat.P[Nat]@(addNat (x#P) y) in
let three = add[Succ[Succ[ε]] two one in
pred three   (* Type: Succ[Succ[Nat]] *)
```

- System F_π의 문법(syntax), 동적 의미(dynamic semantics), 정적 의미(static semantics)를 형식화(formalize) 함

4 한계 및 추후 계획

- 프로그래머가 사용하기에는 매우 복잡함
- 타입 유추(inference)를 통해 동일한 의미의 코드를 간결하게 작성 가능
  - `pred[Nat] (pred[Succ[Nat]] two)` 대신 `pred pred two` 사용
- 타입 유추 알고리즘 설계
- 알고리즘의 올바름과 언어의 타입 안전성(type soundness) 증명

KAIST

본 연구는 과학기술정보통신부 및 정보통신기술진흥센터의 SW중심대학사업의 연구결과로 수행되었음(2016-0-00018)

PLRG Programming Language Research Group