CS 476 – Programming Language Design

William Mansky

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

Lambda Calculus: Types

- The basic ("untyped") lambda calculus has no meaningful types – everything is a function, and any function can be applied to anything as an argument
- But what if we add other kinds of values?

- Next language: lambda calculus with numbers

$\lambda x.\, x + 1$ $\lambda x.\, (\lambda y.\, x + y)$ $\lambda x.\, x\ 5$

Lambda Calculus + Ints

L ::= <ident> | λ<ident>. ***L*** | ***L L*** | <#> | ***L* + *L***

Values are either functions or ints:

4 $\lambda x. 3$ $\lambda x. (\lambda y. x)$ $\lambda x. (\lambda y. (x\ y))$

Exercise: What type should each of these expressions have?

Lambda Calculus + Ints

L ::= <ident> | λ<ident>. ***L*** | ***L L*** | <#> | ***L* + *L***

Values are either functions or ints:

4	$\lambda x.3$	$\lambda x.(\lambda y.x)$	$\lambda x.\left(\lambda y.(x\ y)\right)$
int	int → int	int → (int → int)	(int → int) → (int → int)

Lambda Calculus + Ints

L ::= <ident> | λ<ident>. *L* | *L L* | <#> | *L* **+** *L*

T ::= int | *T* → *T*

Values are either functions or ints:

4	$\lambda x. 3$	$\lambda x. (\lambda y. x)$	$\lambda x. (\lambda y. x\ y)$
int	int → int	int → (int → int)	(int → int) → (int → int)

Simply Typed Lambda Calculus

L ::= <ident> | λ(<ident>: T). L | L L | <#> | L **+** L

T ::= int | $T \rightarrow T$

Values are either functions or ints:

4	$\lambda x\colon \mathrm{int}.\, 3$	$\lambda x\colon \mathrm{int}.\, (\lambda y\colon \mathrm{int}.\, x)$	$\lambda(x\colon \mathrm{int} \rightarrow \mathrm{int}).\, (\lambda y\colon \mathrm{int}.\, x\ y)$
int	int → int	int → (int → int)	(int → int) → (int → int)

Simply Typed Lambda Calculus

L ::= <ident> | λ(<ident>: T). L | L L | <#> | L **+** L

T ::= int | $T \rightarrow T$

- A function with type $A \rightarrow B$ takes type A as input and yields B as output

`bool f(int x){ … }` would have type int → bool

Simply Typed Lambda Calculus

L ::= <ident> | λ(<ident>: T). L | $L\ L$ | <#> | L **+** L

T ::= int | $T \rightarrow T$

- A function with type $A \rightarrow B$ takes type A as input and yields B as output

```
let f (x : int) : bool = …
```

would have type int → bool

Simply Typed Lambda Calculus

L ::= <ident> | λ(<ident>: T). L | L L | <#> | L **+** L

T ::= int | $T \rightarrow T$

- A function with type $A \rightarrow B$ takes type A as input and yields B as output

```
fun (x : int) -> … : bool
```

would have type int → bool

Simply Typed Lambda Calculus

L ::= <ident> | λ(<ident>: *T*). *L* | *L L* | <#> | *L* **+** *L*

T ::= int | *T* → *T*

Values are either functions or ints:

4	$\lambda x\colon \text{int}.\, 3$	$\lambda x\colon \text{int}.\, (\lambda y\colon \text{int}.\, x)$	$\lambda(x\colon \text{int} \rightarrow \text{int}).\, (\lambda y\colon \text{int}.\, x\ y)$
int	int → int	int → (int → int)	(int → int) → (int → int)

Simply Typed Lambda Calculus: Types

L ::= <ident> | *λ*(<ident>: *T*). *L* | *L L* | <#> | *L* **+** *L*

T ::= int | *T* → *T*

$$\frac{(i \text{ is a number literal})}{\Gamma \vdash i : \text{int}}$$

$$\frac{(\Gamma(x) = \tau)}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash l_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash l_2 : \tau_1}{\Gamma \vdash l_1\ l_2 : \tau_2}$$

$$\frac{\Gamma[x \mapsto \tau_1] \vdash l : \tau_2}{\Gamma \vdash (\lambda(x{:}\tau_1).\, l) : \tau_1 \to \tau_2}$$

Simply Typed Lambda Calculus: Types

$$\cfrac{\cfrac{\Gamma[x \mapsto \text{int}] \vdash (\lambda y{:}\,\text{int}.\, x) \;:\; \text{int} \to \text{int}}{\Gamma \vdash (\lambda x{:}\,\text{int}.\,(\lambda y{:}\,\text{int}.\, x)) \;:\; \text{int} \to (\text{int} \to \text{int})} \qquad \Gamma \vdash 4 \;:\; \text{int}}{\Gamma \vdash (\lambda x{:}\,\text{int}.\,(\lambda y{:}\,\text{int}.\, x))\; 4 \;:\; (\text{int} \to \text{int})}$$

$$\frac{\Gamma \vdash l_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash l_2 : \tau_1}{\Gamma \vdash l_1\; l_2 : \tau_2}$$

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

Limitations of Simple Types

- Not every lambda-term is well typed

$$4\ (\lambda x.\, x) \qquad\qquad (\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$$

$$\Gamma \vdash (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) : \tau$$

Limitations of Simple Types

- Not every lambda-term is well typed

$$4\ (\lambda x.\, x) \qquad\qquad (\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$$

$$\frac{\Gamma \vdash (\lambda x.\, x\ x) : \tau_1 \to \tau \quad \Gamma \vdash (\lambda x.\, x\ x) : \tau_1}{\Gamma \vdash (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) : \tau}$$

Limitations of Simple Types

- Not every lambda-term is well typed

$$4\ (\lambda x.\, x) \qquad\qquad (\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$$

$$\frac{\dfrac{\Gamma[x \mapsto \tau_1] \vdash x\ x : \tau}{\Gamma \vdash (\lambda x.\, x\ x) : \tau_1 \to \tau} \qquad \Gamma \vdash (\lambda x.\, x\ x) : \tau_1}{\Gamma \vdash (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) : \tau}$$

Limitations of Simple Types

- Not every lambda-term is well typed

$$4\ (\lambda x.\, x) \qquad\qquad (\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$$

$$\dfrac{\dfrac{\dfrac{\Gamma[x \mapsto \tau_1] \vdash x : \tau_2 \to \tau \quad \Gamma[x \mapsto \tau_1] \vdash x : \tau_2}{\Gamma[x \mapsto \tau_1] \vdash x\ x : \tau}}{\Gamma \vdash (\lambda x.\, x\ x) : \tau_1 \to \tau} \quad \Gamma \vdash (\lambda x.\, x\ x) : \tau_1}{\Gamma \vdash (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) : \tau}$$

Limitations of Simple Types

- Not every lambda-term is well typed

$$4\ (\lambda x. x) \qquad\qquad (\lambda x. x\ x)\ (\lambda x. x\ x)$$

τ_1 can't be the same as $\tau_1 \to \tau$!

$$\dfrac{\dfrac{\dfrac{\Gamma[x \mapsto \tau_1] \vdash x : \tau_1 \to \tau \quad \Gamma[x \mapsto \tau_1] \vdash x : \tau_1}{\Gamma[x \mapsto \tau_1] \vdash x\ x : \tau}}{\Gamma \vdash (\lambda x. x\ x) : \tau_1 \to \tau} \quad \Gamma \vdash (\lambda x. x\ x) : \tau_1}{\Gamma \vdash (\lambda x. x\ x)\ (\lambda x. x\ x) : \tau}$$

Limitations of Simple Types

- Not every lambda-term is well typed

$$4\ (\lambda x.\, x) \qquad\qquad (\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$$

- Untyped lambda terms can run forever, but simply-typed lambda terms always terminate!
  - This means simply-typed lambda calculus is not Turing-complete
  - Many interesting programs (ones that require loops or recursion) can't be written in STLC

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

Limitations of Simple Types

- Not every lambda-term is well typed
- Untyped lambda terms can run forever, but simply-typed lambda terms always terminate!
  - This means simply-typed lambda calculus is not Turing-complete
  - Many interesting programs (ones that require loops or recursion) can't be written in STLC
- Typed languages don't *automatically* include loops/recursion
- But we can add it in as a separate feature

Typed Lambda Calculus with Recursion

L ::= <ident> | λ(<ident>: *T*). *L* | *L L* | <#> | *L* + *L* | *L* − *L*
| `ifzero` *L* `then` *L* `else` *L* | `let rec` <ident> : *T* = *L* `in` *L*

```
let rec f : int -> int =
 λx : int. ifzero x then 1 else x * f (x – 1)
in
  f 5
```

Typed Lambda Calculus with Recursion

L ::= <ident> | λ(<ident>: *T*). *L* | *L* *L* | <#> | *L* **+** *L* | *L* **-** *L*
  | `ifzero` *L* `then` *L* `else` *L* | `let rec` <ident> : *T* = *L* `in` *L*

$$\frac{(i \text{ is a number literal})}{\Gamma \vdash i : \text{int}}$$

$$\frac{(\Gamma(x) = \tau)}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma[x \mapsto \tau_1] \vdash l : \tau_2}{\Gamma \vdash \lambda(x{:}\,\tau_1).\, l : \tau_1 \to \tau_2}$$

$$\frac{?}{\Gamma \vdash (\texttt{let rec}\ x : \tau = l_1\ \texttt{in}\ l_2) : \tau_2}$$

- Exercise: How would you typecheck a **`let rec`**?

Typed Lambda Calculus with Recursion

L ::= <ident> | *λ*(<ident>: *T*). *L* | *L* *L* | <#> | *L* **+** *L* | *L* **−** *L*

| `ifzero` *L* `then` *L* `else` *L* | `let rec` <ident> : *T* = *L* `in` *L*

$$\frac{(i \text{ is a number literal})}{\Gamma \vdash i : \text{int}}$$

$$\frac{(\Gamma(x) = \tau)}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma[x \mapsto \tau_1] \vdash l : \tau_2}{\Gamma \vdash \lambda(x{:}\,\tau_1).\, l : \tau_1 \to \tau_2}$$

$$\frac{\Gamma[x \mapsto \tau] \vdash l_2 : \tau_2}{\Gamma \vdash (\texttt{let rec}\ x : \tau = l_1\ \texttt{in}\ l_2) : \tau_2}$$

Typed Lambda Calculus with Recursion

L ::= <ident> | λ(<ident>: *T*). *L* | *L L* | <#> | *L* + *L* | *L* − *L*

| `ifzero` *L* `then` *L* `else` *L* | `let rec` <ident> : *T* = *L* `in` *L*

$$\frac{(i \text{ is a number literal})}{\Gamma \vdash i : \text{int}}$$

$$\frac{(\Gamma(x) = \tau)}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma[x \mapsto \tau_1] \vdash l : \tau_2}{\Gamma \vdash \lambda(x{:}\,\tau_1).\, l : \tau_1 \to \tau_2}$$

$$\frac{\Gamma[x \mapsto \tau] \vdash l_1 : \tau \quad \Gamma[x \mapsto \tau] \vdash l_2 : \tau_2}{\Gamma \vdash (\texttt{let rec}\ x : \tau = l_1\ \texttt{in}\ l_2) : \tau_2}$$

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

Typed Lambda Calculus with Recursion

L ::= <ident> | λ(<ident>: *T*). *L* | *L L* | <#> | *L* **+** *L* | *L* **−** *L*
| `ifzero` *L* `then` *L* `else` *L* | `let rec` <ident> : *T* = *L* `in` *L*

$$\frac{l_1 \to l_1'}{l_1\ l_2 \to l_1'\ l_2}$$

$$\frac{}{(\lambda(x{:}\,\tau).\,l)\ v \to [x \mapsto v]l}$$

$$\frac{l_2 \to l_2'}{v\ l_2 \to v\ l_2'}$$

$$\frac{}{\texttt{let rec}\ x : \tau = l_1\ \texttt{in}\ l_2 \to ?}$$

Typed Lambda Calculus with Recursion

L ::= <ident> | λ(<ident>: *T*). *L* | *L* *L* | <#> | *L* **+** *L* | *L* **−** *L*

| `ifzero` *L* `then` *L* `else` *L* | `let rec` <ident> : *T* = *L* `in` *L*

```
let rec f = λx. ifzero x then 1 else x*f (x−1) in
  f 5 →
```

```
(λx. ifzero x then 1 else x * f (x−1)) 5
```

Typed Lambda Calculus with Recursion

L ::= <ident> | λ(<ident>: *T*). *L* | *L* *L* | <#> | *L* **+** *L* | *L* **−** *L*
  | `ifzero` *L* `then` *L* `else` *L* | `let rec` <ident> : *T* = *L* `in` *L*

```
let rec f = λx. ifzero x then 1 else x*f (x-1) in
  f 5 →
```

```
(λx. ifzero x then 1 else x * f (x-1)) 5 → … →
5 * f 4
```

But what is `f`?

Typed Lambda Calculus with Recursion

L ::= <ident> | λ(<ident>: *T*). *L* | *L L* | <#> | *L* **+** *L* | *L* **−** *L*

| `ifzero` *L* `then` *L* `else` *L* | `let rec` <ident> : *T* = *L* `in` *L*

```
let rec f = λx. ifzero x then 1 else x*f (x-1)
  in f 5 →
let rec f = λx. ifzero x then 1 else x*f (x-1)
  in (λx. ifzero x then 1 else x*f (x-1)) 5
```

Typed Lambda Calculus with Recursion

L ::= <ident> | λ(<ident>: *T*). *L* | *L* *L* | <#> | *L* **+** *L* | *L* **−** *L*

| `ifzero` *L* `then` *L* `else` *L* | `let rec` <ident> : *T* = *L* `in` *L*

```
let rec f = λx. ifzero x then 1 else x*f (x-1)
  in (λx. ifzero x then 1 else x*f (x-1)) 5
```

Typed Lambda Calculus with Recursion

L ::= <ident> | λ(<ident>: *T*). *L* | *L* *L* | <#> | *L* **+** *L* | *L* **-** *L*

| `ifzero` *L* `then` *L* `else` *L* | `let rec` <ident> : *T* = *L* `in` *L*

```
let rec f = λx. ifzero x then 1 else x*f (x-1)
  in 5 * f 4
```

Typed Lambda Calculus with Recursion

L ::= <ident> | λ(<ident>: *T*). *L* | *L L* | <#> | *L* **+** *L* | *L* **−** *L*

| `ifzero` *L* `then` *L* `else` *L* | `let rec` <ident> : *T* = *L* `in` *L*

```
let rec f = λx. ifzero x then 1 else x*f (x–1)
  in 5 * (λx. ifzero x then 1 else x*f (x–1)) 4
```

Typed Lambda Calculus with Recursion

L ::= <ident> | λ(<ident>: *T*). *L* | *L L* | <#> | *L* **+** *L* | *L* **−** *L*

| `ifzero` *L* `then` *L* `else` *L* | `let rec` <ident> : *T* = *L* `in` *L*

```
let rec f = λx. ifzero x then 1 else x*f (x-1)
  in (* after many steps *) 120
```

→ `120`

Typed Lambda Calculus with Recursion

L ::= <ident> | *λ*(<ident>: *T*). *L* | *L L* | <#> | *L* **+** *L* | *L* **-** *L*
| `ifzero` *L* `then` *L* `else` *L* | `let rec` <ident> : *T* = *L* `in` *L*

$$\frac{}{\texttt{let rec}\ x : \tau = l_1\ \texttt{in}\ l_2 \to \texttt{let rec}\ x : \tau = l_1\ \texttt{in}\ [x \mapsto l_1]l_2}$$

$$\frac{l_2 \to l_2'}{\texttt{let rec}\ x : \tau = l_1\ \texttt{in}\ l_2 \to \texttt{let rec}\ x : \tau = l_1\ \texttt{in}\ l_2'}$$

$$\frac{}{\texttt{let rec}\ x : \tau = l_1\ \texttt{in}\ v \to v}$$

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.