Simply typed lambda calculus

Add types and static typechecking to lambda calculus

"Simply typed": no polymorphic types

```
E      ::= λI:τ.E
         | E E
         | I
τ      ::= *
         | τ → τ
```

[Syntactic associativity rules:
arrow is right-associative: $\tau_1 \to \tau_2 \to \tau_3 = \tau_1 \to (\tau_2 \to \tau_3)$]

α, β, η-rules apply to this calculus, too
(α-rule must rename with identifier having same type)

- confluence, normalization, standardization properties apply too

Typechecking rules

A **typing environment** Γ is a mapping from identifiers to types, giving for each identifier in scope its corresponding type

A **typing judgment** that
an expression E has type τ in typing environment Γ is written
$\Gamma \vdash E : \tau$

Typechecking rules for language given in **syntax-directed** way:
for each syntactic construct, give **inference rule**(**s**) for all legal ways of computing that construct's type

Write inference rules in "natural deduction" style

Rules for simply typed lambda calculus:

[var] $\quad \Gamma \vdash I{:}\tau \quad$ if $I{:}\tau \in \Gamma$

[$\to$ intro] $\quad \dfrac{\Gamma \cup I{:}\tau_1 \vdash E{:}\tau_2}{\Gamma \vdash (\lambda I{:}\tau_1.E){:}\tau_1 \to \tau_2}$

[$\to$ elim] $\quad \dfrac{\Gamma \vdash E_1{:}\tau_1 \to \tau_2 \qquad \Gamma \vdash E_2{:}\tau_1}{\Gamma \vdash (E_1\ E_2){:}\tau_2}$

Properties of simply typed lambda calculus

Uniqueness of typing:
only one way to assign types successfully

Strong normalization:
all programs have a normal form (!)

Adding a fixpoint operator

Can get the effect of a recursive definition using the **paradoxical Y combinator** a.k.a. the fixpoint operator

Definition (in untyped lambda calculus):

```
fix ≡ λF.((λI. F (I I)) (λI. F (I I)))
```

But cannot type `fix`'s definition, and it's polymorphic, so build it in as a primitive:

$\texttt{fix}_\tau : (\tau \to \tau) \to \tau$

Example use, in typed lambda calculus:

```
nat ≡ (*→*)→*→*
fact ≡ fix_nat→nat (λfact:nat→nat.
  λn:nat. if (eq n 0) 1 (mul n (fact (sub n 1))))
```

Polymorphic lambda calculus

(A.k.a. second-order lambda calculus)

Add “functions” taking types as arguments,
applications of these “functions” to types

```
E    ::= λI:τ.E
       | E E
       | ΛI.E
       | E τ
       | I
τ    ::= *
       | τ → τ
       | ∀I.τ
       | I
```

Additional β-rule, -reduction for type application:

$(\Lambda \mathtt{I.E})\tau \Leftrightarrow [\tau/\mathtt{I}]\mathtt{E}$

Some examples:

```
Id ≡ (Λt.λx:t.x) : ∀t.t→t
Id nat ⇒ (λx:nat.x) : nat→nat
Id (nat→nat) ⇒
  (λx:nat→nat.x) : (nat→nat)→(nat→nat)
Id (nat→nat) (Id nat) 3
```


Typing rules

[var] $\Gamma \vdash \mathtt{I}:\tau$ if $\mathtt{I}:\tau \in \Gamma$

[→ intro] $$\frac{\Gamma \cup \mathtt{I}:\tau_1 \vdash \mathtt{E}:\tau_2 \qquad \Gamma \vdash \tau_1 : \text{type} \qquad \Gamma \vdash \tau_2 : \text{type}}{\Gamma \vdash (\lambda \mathtt{I}:\tau_1.\mathtt{E}):\tau_1 \to \tau_2}$$

[→ elim] $$\frac{\Gamma \vdash \mathtt{E}_1:\tau_1 \to \tau_2 \qquad \Gamma \vdash \mathtt{E}_2:\tau_1}{\Gamma \vdash (\mathtt{E}_1\ \mathtt{E}_2):\tau_2}$$

[∀ intro] $$\frac{\Gamma \cup \mathtt{I}:\text{type} \vdash \mathtt{E}:\tau}{\Gamma \vdash (\Lambda \mathtt{I}.\mathtt{E}):\forall \mathtt{I}.\tau}$$

[∀ elim] $$\frac{\Gamma \vdash \mathtt{E}:\forall \mathtt{I}.\tau_2 \qquad \Gamma \vdash \tau_1 : \text{type}}{\Gamma \vdash (\mathtt{E}\ \tau_1):[\tau_1/\mathtt{I}]\tau_2}$$

What’s a legal argument type for a regular λ?

Predicative polymorphic lambda calculus: just regular types

[typebase] $\Gamma \vdash *:\text{type}$

[typevar] $\Gamma \vdash \mathtt{I}:\text{type}$ if $\mathtt{I}:\text{type} \in \Gamma$

[→] $$\frac{\Gamma \vdash \tau_1 : \text{type} \qquad \Gamma \vdash \tau_2 : \text{type}}{\Gamma \vdash \tau_1 \to \tau_2 : \text{type}}$$

Impredicative polymorphic lambda calculus:
also allow polymorphic values

Add

[∀] $$\frac{\Gamma \cup \mathtt{I}:\text{type} \vdash \tau : \text{type}}{\Gamma \vdash (\forall \mathtt{I}.\tau) : \text{type}}$$

- [Girard 71] System F

Comparison

Predicative is simple to think about

- functions have only values/functions of simpler type as arguments, results
  - domain/range of function does not include itself
  - can use intuitive set-theoretic model of functions (fn = set of input/output ordered pairs)
- [Martin-Löf 84] constructive/intuitionistic type theory

Impredicative is powerful

- allows functions to take polymorphic arguments, since an argument type may be polymorphic
- polymorphic functions can be passed themselves as arguments
  - includes itself in its domain/range
  - set-theoretic model of functions leads to Russell’s paradox
- type inference is undecidable

Let-bound polymorphism *á la* ML is
predicative + `let` declaration

- can’t use lambda’s for naming polymorphic fns

All are strongly normalizing....

An example

OK in predicative and impredicative:

```
Compose ≡ Λa.Λb.Λc.λf:b→c.λg:a→b.λx:a.f (g x)
Compose nat nat nat (Id nat) (Id nat) 3
```

- ≡ definitions & expansions analogous to ML's `let` w/o `fix`

OK only in impredicative:
  `λf`... returns a function of polymorphic type

```
Compose2 ≡ Λb.Λc.λf:b→c.Λa.λg:a→b.λx:a.f g x
C ≡ Compose2 nat nat (Id nat)
    : ∀a.(a→nat)→nat→nat
C nat (Id nat) 3
apply-to-1 ≡ λf:nat->nat. f 1
C (nat->nat) apply-to-1 (add 3)
```

OK only in impredicative:
  instantiating with a polymorphic type

```
Id ≡ (Λt.λx:t.x) : ∀t.t→t
Id (∀t.t→t) Id : ∀t.t→t
```


A third type system: type:type

Types can be constructed from arbitrary expressions:

```
τ    ::= ...
       | E
```

Expressions that yield types are legal
  function arguments/results:

[typetype] $\Gamma \vdash \text{type:type}$

Properties:

- types and values become unified!
- λ and Λ become the same!
- can do arbitrary computations at type-expression level!
- type checking becomes undecidable....