Simply typed λ-calculus

Add types and static typechecking to λ-calculus

- "simply typed": no polymorphic types

Syntax: add types to formals:

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

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

$*$ is a base type, for values that will never be called

Typing environments and judgments

Typing environment Γ: a sequence of $I:\tau$ pairs

- records the type of each bound identifier
- e.g.: x:*,y:*→*,z:*
- empty sequence: ∅

Typing judgment of the form $\Gamma \vdash E : \tau$

- "in typing environment Γ,
  syntactically well-formed expression E is also
  semantically well-formed, and has type τ"
- $\vdash$ and : are just punctuation; could have been (Γ, E, τ)

A (correct) typing judgment:
x:*,y:*→*,z:* ⊢ (y z) : *

An (incorrect) typing judgment:
x:*,y:*→*,z:* ⊢ (w (z y)) : *→*

Static semantics:
a set of rules that specify which typing judgments are correct

Inference rules

Can specify a set of legal judgments using a collection of logical **inference rules** of the following form:

$$\frac{\text{premise}_1 \quad \ldots \quad \text{premise}_k}{\text{conclusion}} \qquad (k \geq 0)$$

- whenever all the premises are true, the conclusion is true
- a rule with no premises is an **axiom**
- rules can have "side conditions" that constrain when they apply

Example rule: $\dfrac{A \Rightarrow B \qquad A}{B}$

Premises and conclusions can containing meta-variables

- instantiate meta-variables consistently within a rule

Constructive: something is in the set of facts being specified only if it can be deduced from axioms by applying instantiations of inference rules a finite number of times

- if something can't be deduced, then it's not in the set

Static semantics inference rules

Specified in a **syntax-directed** way:
for each syntactic construct, give **inference rule**(**s**) for all ways of that construct is well-formed

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

[→ intro] $\dfrac{\Gamma, \mathtt{I}:\tau_1 \vdash \mathtt{E}:\tau_2}{\Gamma \vdash (\lambda \mathtt{I}:\tau_1.\mathtt{E}):\tau_1 \to \tau_2}$ if $\mathtt{I}:\tau \notin \Gamma$

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

A program E is semantically well-formed iff $\varnothing \vdash E : \tau$ is derivable

- statically illegal programs are specified by *omission*

That's it!

Typing derivations

To demonstrate (a.k.a. prove) that
an expression *E* has type τ in typing environment Γ,
provide a **typing derivation**

- a tree of instances of typing inference rules,
where the conclusion of one rule is a premise of the next,
whose leaves are axiom instances and whose final conclusion is $\Gamma \vdash E : \tau$

Specification vs. algorithm

Static semantic rules are a *specification*: don't say how to check whether a program is correct, just say how to verify a supposed proof that a program is

- oracles are OK!

Real type checkers require a type checking *algorithm* that will compute whether a program is type-correct

- no oracles
- termination is good!

Can read many inference rules as if they were cases in an algorithm

- Γ and *E* in conclusion as "inputs"
- recursively typecheck subexpressions, augmenting Γ if needed, to compute their result types
- compute and return conclusion's result type as "output"

$$[\to \text{intro}] \qquad \frac{\Gamma, \mathtt{I}:\tau_1 \vdash \mathtt{E}:\tau_2}{\Gamma \vdash (\lambda \mathtt{I}:\tau_1.\mathtt{E}):\tau_1 \to \tau_2} \qquad \text{if } \mathtt{I}:\tau \notin \Gamma$$

An alternative language

Syntax: same, but *omit* explicit type of formal argument

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

Typing rules: same, but *infer* type of λ formal

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

$$[\to \text{intro}] \qquad \frac{\Gamma, \mathtt{I}:\tau_1 \vdash \mathtt{E}:\tau_2}{\Gamma \vdash (\lambda \mathtt{I}.\mathtt{E}):\tau_1 \to \tau_2} \qquad \text{if } \mathtt{I}:\tau \notin \Gamma$$

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

A fine specification, but it's trickier to implement...

- [is it impossible to implement?]

["Church-style" vs. "Curry-style"]

Specifying evaluation

Can specify evaluation rules precisely using inference rules, too

Judgments of the form $E_1 \to E_2$

- "expression E_1 reduces in one step to E_2"

Can formalize different reduction semantics
E.g., full reduction:

$$[\beta] \qquad \frac{}{(\lambda \mathtt{I}:\tau.\mathtt{E}_1)\mathtt{E}_2 \to [\mathtt{E}_2/\mathtt{I}]\mathtt{E}_1}$$

$$[\lambda] \qquad \frac{\mathtt{E} \to \mathtt{E}'}{\lambda \mathtt{I}:\tau.\mathtt{E} \to \lambda \mathtt{I}:\tau.\mathtt{E}'}$$

$$[\text{app}_1] \qquad \frac{\mathtt{E}_1 \to \mathtt{E}_1'}{\mathtt{E}_1\ \mathtt{E}_2 \to \mathtt{E}_1'\ \mathtt{E}_2}$$

$$[\text{app}_2] \qquad \frac{\mathtt{E}_2 \to \mathtt{E}_2'}{\mathtt{E}_1\ \mathtt{E}_2 \to \mathtt{E}_1\ \mathtt{E}_2'}$$

[How to specify normal order? call-by-name? call-by-value?]

[How to specify $\to^*$?]