Simply typed $\lambda$-calculus

Add types and static typechecking to $\lambda$-calculus

• “simply typed”: no polymorphic types

Syntax: add types to formals:

$$
E ::= \lambda I : \tau. E \\
E E \\
I
$$

\[\tau ::= * \\
\tau \rightarrow \tau\]

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

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

Typing environments and judgments

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

• records the type of each bound identifier
• e.g.: $x : * , y : * \rightarrow * , z : *$
• empty sequence: $\emptyset$

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

• “in typing environment $\Gamma$,
syntactically well-formed expression $E$ is also
semantically well-formed, and has type $\tau$”.
• $\vdash$ and $: \text{are just punctuation; could have been } (\Gamma, E, \tau)$

A (correct) typing judgment:

$$x : *, y : * \rightarrow *, z : * \vdash (y \ z) : *$$

An (incorrect) typing judgment:

$$x : *, y : * \rightarrow *, z : * \vdash (w \ (z \ y)) : * \rightarrow *$$

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:

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

• 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: $A \Rightarrow B \quad A \quad \frac{}{B}$

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

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

\[\text{[\rightarrow intro]} \quad \frac{\Gamma, I : \tau_1 \vdash E : \tau_2}{\Gamma \vdash (\lambda I : \tau_1. E) : \tau_1 \rightarrow \tau_2} \quad \text{if } I : \tau \notin \Gamma\]

\[\text{[\rightarrow elim]} \quad \frac{\Gamma \vdash E_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash E_2 : \tau_1}{\Gamma \vdash (E_1 \ E_2) : \tau_2} \]

A program $E$ is semantically well-formed iff $\emptyset \vdash E : \tau$ 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 $\tau$ in typing environment $\Gamma$, 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

• $\Gamma$ and $E$ in conclusion as “inputs”
• recursively typecheck subexpressions, augmenting $\Gamma$ if needed, to compute their result types
• compute and return conclusion’s result type as “output”

\[
\Gamma, I : \tau_1 \vdash E : \tau_2 \\
\Gamma \vdash (\lambda I : \tau_1. E) : \tau_1 \rightarrow \tau_2
\] if $I : \tau \not\in \Gamma$

An alternative language

Syntax: same, but omit explicit type of formal argument

\[
E :: = \lambda I . \ E \\
| \ E \ E \\
| I \\
\tau :: = * \\
| \tau \rightarrow \tau
\]

Typing rules: same, but infer type of $\lambda$ formal

\[
\frac{}{\Gamma \vdash I : \tau} \quad \text{[var]} \\
\frac{\Gamma, I : \tau_1 \vdash E : \tau_2 \quad \text{if } I : \tau \in \Gamma}{\Gamma \vdash (\lambda I . E) : \tau_1 \rightarrow \tau_2} \quad \text{[\rightarrow intro]} \\
\frac{\Gamma \vdash E_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash E_2 : \tau_1}{\Gamma \vdash (E_1 \ E_2) : \tau_2} \quad \text{[\rightarrow elim]}
\]

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 $E_1 \rightarrow E_2$

• “expression $E_1$ reduces in one step to $E_2$”

Can formalize different reduction semantics

E.g., full reduction:

\[
[\beta] \quad \frac{(\lambda I : \tau. E_1) E_2 \rightarrow [E_2/I] E_1}{E_1 \rightarrow E_1'} \\
[\lambda] \quad \frac{E \rightarrow E'}{\lambda I : \tau. E ightarrow \lambda I : \tau. E'} \\
[\text{app}_1] \quad \frac{E_1 \ E_2 \rightarrow E_1 \ E_2'}{E_1 \rightarrow E_1'} \\
[\text{app}_2] \quad \frac{E_1 \ E_2 \rightarrow E_1 \ E_2'}{E_2 \rightarrow E_2'}
\]

[How to specify normal order? call-by-name? call-by-value?] [How to specify $\rightarrow^*$?]