De-sugared Imperative Language

- Syntax:

\[ E ::= E_1 := E_2 \mid E_1; E_2 \mid \text{if } E \text{ then } E_1 \text{ else } E_2 \text{ fi} \mid \text{while } E \text{ do } E_1 \text{ od} \]

\[ \mid \text{skip} \mid \text{newint} \mid E_1 + E_2 \mid E_1 = E_2 \mid \neg E \mid \@E \mid N \mid L \]

- ... plus
  1. type tags
  2. typing rules
  3. records and lambda abstractions
Type Attributes and Meanings

**Primitive:** \( int, bool, store \)

\[
\left[\text{int}\right] = \mathbb{Z}_{\perp}, \left[\text{bool}\right] = \{\text{true, false}\}_{\perp}
\]
\[
\left[\text{store}\right] = \{\langle n_1, \ldots, n_k \rangle \mid n_i \in \mathbb{Z}\}_{\perp}
\]

**Structured:** \( \theta \rightarrow \theta', \{l : \theta_l\}_{l \in \mathcal{I}} \)

\[
\left[\theta \rightarrow \theta'\right] = \left[\theta\right] \rightarrow \left[\theta'\right]
\]
\[
\left[\{l : \theta_l\}_{l \in \mathcal{I}}\right] = \prod_{l \in \mathcal{I}} \left[\theta_l\right]
\]

**Unevaluated:** \( \theta^\text{exp} \)

\[
\left[\theta^\text{exp}\right] = \text{Store} \rightarrow \left[\theta\right]
\]
Typing Rules and vs. Semantics

\[\text{succ} : \text{int} \rightarrow \text{int} \quad \text{e} : \text{intexp}\]

\[
\begin{array}{c}
\hline
\text{succ} (\text{e}) : \text{?} \\
\hline
\end{array}
\]

Eager: \[\llbracket \text{succ} (\text{e}) \rrbracket s = (\llbracket \text{e} \rrbracket s) + 1 : \text{int}\]

Lazy: \[\llbracket \text{succ} (\text{e}) \rrbracket s = \lambda s. (\llbracket \text{e} \rrbracket s) + 1 : \text{intexp}\]

... how do we decide which one is “right”?  
⇒ pick one “way” (pretty much everywhere)  
⇒ add syntax that makes the choice explicit
Idea

The Eager/Lazy difference doesn’t show up until we can “move” pieces of code around (by naming them).

⇒ hence the issues are usually attached to naming devices.

- identifier bindings are explicitly declared eager/lazy
  How?? type tags! (\(\theta\) vs. \(\theta_{exp}\))
- an alternative: explicit eval keyword
  with following typing and semantics:

\[
\begin{align*}
  e : \theta_{exp} \\
  \text{eval } e : \theta \\
  [\text{eval } e]s = [e]s
\end{align*}
\]
Higher-order Languages

Idea

*We allow records and functions to be values.*

- values of parameters/return values of functions
- components of records
Summary

- Imperative language = *While*-core+records+parameters
  - centered around operations on *Stores*
  - identifiers are typically *statically scoped*
- *Stores* are *implicit* parameters to all *expressions*
  - lazy expressions just “ignore it”
- Lazy/Eager-ness of expressions
  - commonly determined when they are *bound to an identifier*.
- Questions:
  1. Can pure lazy evaluation “simulate” eager constructs?
     Can pure eager evaluation “simulate” lazy constructs?
  2. Can *λ*-abstractions simulate records?
  3. Do we really need *Store*
     (now that we can create records/functions)?
  4. How is the Eager/Lazy issue impacted by *recursive* abstractions?