Records and Lambda Abstractions Revisited

Programming Languages CS442

David Toman

School of Computer Science
University of Waterloo

De-sugared Imperative Language

- Syntax:

$$E \quad ::= \quad E_1 := E_2 \mid E_1; E_2 \mid \textbf{if}\, E\, \textbf{then}\, E_1\, \textbf{else}\, E_2\, \textbf{fi} \mid \textbf{while}\, E\, \textbf{do}\, E_1\, \textbf{od}$$
$$\mid \textbf{skip} \mid \textbf{newint} \mid E_1 + E_2 \mid E_1 = E_2 \mid \neg E \mid @E \mid N \mid L$$

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

Type Attributes and Meanings

Primitive: $int, bool, store$

$$[\![int]\!] = Z_\perp, [\![bool]\!] = \{true, false\}_\perp$$
$$[\![store]\!] = \{\langle n_1, \ldots, n_k \rangle \mid n_i \in Z\}_\perp$$

Structured: $\theta \to \theta', \{I : \theta_I\}_{I \in \mathcal{I}}$

$$[\![\theta \to \theta']\!] = [\![\theta]\!] \to [\![\theta']\!]$$
$$[\![\{I : \theta_I\}_{I \in \mathcal{I}}]\!] = \prod_{I \in \mathcal{I}} [\![\theta_I]\!]$$

Unevaluated: θexp

$$[\![\theta exp]\!] = Store \to [\![\theta]\!]$$

Typing Rules and/vs. Semantics

$$\frac{succ : int \rightarrow int \qquad e : intexp}{succ(e) : ?}$$

Eager: $[\![succ(e)]\!]s = ([\![e]\!]s) + 1 : int$

Lazy: $[\![succ(e)]\!]s = \lambda s.([\![e]\!]s) + 1 : intexp$

. . . how do we decide which one is “right”?

$\Rightarrow$ pick one “way” (pretty much everywhere)

$\Rightarrow$ add syntax that makes the choice explicit

Records and Abstractions

Idea

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

$\Rightarrow$ hence the issues are usually attached to naming devices.

- identifier bindings are *explicitly* declared eager/lazy

  How?? type tags! (θ vs. θexp)

- an alternative: explicit **eval** keyword

  with following typing and semantics:

$$\frac{e : \theta exp}{\textbf{eval}\, e : \theta} \qquad [\![\textbf{eval}\, e]\!]s = [\![e]\!]s$$

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

  $\Rightarrow$ centered around operations on *Stores*

  $\Rightarrow$ identifiers are typically *statically scoped*

- *Stores* are *implicit* parameters to all *expressions*

  $\Rightarrow$ 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?