Records and Lambda Abstractions

Project Seminar "Formal Methods I" –
Sub-Seminar "Type Systems"

Ana Carević

Johannes Kepler University, Linz, Austria

The "Desugared" Programming Language

- The "Desugared" Core Imperative Programming Language
  - rewritten syntax definition: all the constructions reside within the same syntax rule

$E \in$ Everything

$N \in$ Numeral

$L \in$ Location

$$E ::= E_1 := E_2 \mid E_1 ; E_2 \mid \textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3 \textbf{ fi} \mid \textbf{skip}$$
$$\mid \textbf{while } E_1 \textbf{ do } E_2 \textbf{ od} \mid E_1 + E_2 \mid E_1 = E_2 \mid \neg E \mid @E \mid N \mid L$$

Before:

- A Core Imperative Language
  - distinct syntax domains

$C \in \text{Command}$

$E \in \text{Expression}$

$L \in \text{Location}$

$N \in \text{Numeral}$

$C ::= L{:=}E \mid C_1;C_2 \mid \textbf{if } E \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}$
$\quad \mid \textbf{while } E \textbf{ do } C \textbf{ od} \mid \textbf{skip}$

$E ::= N \mid @L \mid E_1{+}E_2 \mid \neg E \mid E_1{=}E_2$

$L ::= \mathit{loc}_i$, if $i > 0$

$N ::= n$, if $n \in \mathit{Integer}$

The "Desugared" Programming Language

- Typing Rules
  - typing annotations preserve their structure

$$\frac{E_1 : \mathit{intloc} \qquad E_2 : \mathit{intexp}}{E_1 := E_2 : \mathit{comm}} \qquad \frac{E_1 : \mathit{comm} \qquad E_2 : \mathit{comm}}{E_1 ; E_2 : \mathit{comm}}$$

$$\frac{E_1 : \mathit{boolexp} \qquad E_2 : \mathit{comm} \qquad E_3 : \mathit{comm}}{\textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3 \textbf{ fi} : \mathit{comm}} \qquad \textbf{skip} : \mathit{comm} \qquad \frac{E_1 : \mathit{boolexp} \qquad E_2 : \mathit{comm}}{\textbf{while } E_1 \textbf{ do } E_2 \textbf{ od} : \mathit{comm}}$$

$$\frac{E_1 : \mathit{intexp} \qquad E_2 : \mathit{intexp}}{E_1 + E_2 : \mathit{boolexp}} \qquad \frac{E_1 : \tau\ \mathit{exp} \qquad E_2 : \tau\ \mathit{exp}}{E_1 = E_2 : \mathit{intexp}} \qquad \frac{E : \mathit{boolexp}}{\neg\, E : \mathit{boolexp}}$$

$$\frac{E : \mathit{intloc}}{@E : \mathit{intexp}} \qquad N : \mathit{intexp} \qquad L : \mathit{intloc}$$

Before:

- Typing Rules

Command

$$\frac{\mathsf{L}: \mathit{intloc} \quad \mathsf{E}: \mathit{intexp}}{\mathsf{L}{:=}\mathsf{E}: \mathit{comm}} \qquad \frac{\mathsf{C}_1: \mathit{comm} \quad \mathsf{C}_2: \mathit{comm}}{\mathsf{C}_1;\mathsf{C}_2: \mathit{comm}}$$

$$\frac{\mathsf{E}: \mathit{boolexp} \quad \mathsf{C}_1: \mathit{comm} \quad \mathsf{C}_2: \mathit{comm}}{\textbf{if } \mathsf{E} \textbf{ then } \mathsf{C}_1 \textbf{ else } \mathsf{C}_2 \textbf{ fi}: \mathit{comm}}$$

$$\frac{\mathsf{E}: \mathit{boolexp} \qquad \mathsf{C}: \mathit{comm}}{\textbf{while } \mathsf{E} \textbf{ do } \mathsf{C} \textbf{ od}: \mathit{comm}} \quad \textbf{skip}: \mathit{comm}$$

Expression

$$\frac{\mathsf{N}: \mathit{int}}{\mathsf{N}: \mathit{intexp}} \qquad \frac{\mathsf{L}: \mathit{intloc}}{@\mathsf{L}: \mathit{intexp}}$$

$$\frac{\mathsf{E}_1: \mathit{intexp} \quad \mathsf{E}_2: \mathit{intexp}}{\mathsf{E}_1{+}\mathsf{E}_2: \mathit{intexp}} \qquad \frac{\mathsf{E}: \mathit{boolexp}}{\neg\mathsf{E}: \mathit{boolexp}}$$

$$\frac{\mathsf{E}_1: \tau\mathit{exp} \quad \mathsf{E}_2: \tau\mathit{exp}}{\mathsf{E}_1{=}\mathsf{E}_2: \mathit{boolexp}} \quad \text{if } \tau \in \{\mathit{int}, \mathit{bool}\}$$

Location

loc_i: *intloc*, if $i > 0$

Numeral

n: *int*, if $n \in$ *Integer*

Record Introduction

- Applying the abstraction / qualification principle
- Augmenting the "desugared" programming language:

$$E ::= \cdots \mid I = E \mid E_1, E_2 \mid \mathit{with}\, E_1\, \mathit{do}\, E_2 \mid I$$

$$\frac{\pi \vdash E : \Theta}{\pi \vdash I = E : \{I : \Theta\}} \qquad \frac{\pi \vdash E_1 : \pi_1 \quad \pi \vdash E_2 : \pi_2}{\pi \vdash E_1, E_2 : \pi_1 \dot{\cup} \pi_2}$$

$$\frac{\pi \vdash E_1 : \pi_1 \quad \pi \,\overline{\cup}\, \pi_1 \vdash E_2 : \Theta}{\pi \vdash \mathit{with}\, E_1\, \mathit{do}\, E_2 : \Theta} \qquad \pi \vdash I : \Theta \quad \text{if} \quad (I : \Theta) \in \pi$$

Record Introduction

- I = E → *record*
- E_1, E_2 → unions two records together
- *with* E_1 *do* E_2 → the "desugared" form of **begin** E_1 **in** E_2 **end**
- I → refers to a field in a record

Record Introduction

• The record construction allows phrases from all syntax domains to be named → the abstraction principle

•The *with* construction ensures that phrases from all syntax domains may have local definitions → the qualification principle

• The introduction of records enriches the type system of the language:

- the type $\{i : \theta_i\}_{i \in I}$ is used as the type attribute for records
- phrases can include identifiers → the typing of the phrases requires a type assignment attribute
- all of the typing rules are enriched in the expected way, e.g.

$$\frac{\pi \vdash E_1 : \mathit{intloc} \qquad \pi \vdash E_2 : \mathit{int\ exp}}{\pi \vdash E_1 := E_2 : \mathit{comm}}$$

Lambda Abstraction Introduction

• Applying the parameterization principle by augmenting the language with these new constructions and typing rules:

$$E ::= \cdots \mid \lambda I : \Theta . E \mid E_1 E_2 \mid I$$

$$\frac{\pi \cup \{I : \Theta_1\} \vdash E : \Theta_2}{\pi \vdash \lambda I : \Theta_1 . E : \Theta_1 \to \Theta_2} \qquad \frac{\pi \vdash E_1 : \Theta_1 \to \Theta_2 \qquad \pi \vdash E_2 : \Theta_1}{\pi \vdash E_1 E_2 : \Theta_2}$$

$$\pi \vdash I : \Theta, \text{ if } (I : \Theta) \in \pi$$

Lambda Abstraction Introduction

- The lambda abstraction $\lambda I_2 : \theta . E \rightarrow$ the "desugared" body of a parameterized abstraction **define** $I_1(I_2 : \theta) = E$
- $E_1 E_2 \rightarrow$ the "desugared" form of abstraction invocation with an actual parameter
- $E_1 \rightarrow$ invoked abstraction
- $I \rightarrow$ parameter reference

Lambda Abstraction Introduction

• The lambda abstraction allows phrases from all syntax domains to be parameters to phrases → the parameterization principle

• The introduction of parameters enriches the type system of the language:

- the type $\Theta_1 \rightarrow \Theta_2$ is used as the type attribute for lambda abstractions
- the same identifier construct I used for both record identifiers and parameter identifiers → the lone type assignment π suffices to resolve identifier references

Higher-Order Programming Languages

- The imperative language with records and lambda abstraction:

$E ::= E_1 := E_2 \mid E_1;E_2 \mid \textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3 \textbf{ fi} \mid \textbf{skip} \mid \textbf{while } E_1 \textbf{ do } E_2 \textbf{ od}$

$\mid E_1 + E_2 \mid E_1 = E_2 \mid \neg E \mid @E \mid N \mid L$

$\mid I = E \mid E_1, E_2 \mid \mathit{with}\, E_1\ \mathit{do}\ E_2 \mid I \mid \lambda I : \Theta . E \mid E_1 E_2$

$\Theta ::= \tau\ \mathit{exp} \mid \mathit{comm} \mid \mathit{intloc} \mid \pi \mid \Theta_1 \rightarrow \Theta_2$

$\tau ::= \mathit{int} \mid \mathit{bool}$

$\pi ::= \{j : \Theta_j\}_{j \in J}$, where $J \subseteq$ Identifier is a finite set

Higher-Order Programming Languages

- The imperative language with records and lambda abstraction:

$$\frac{\pi \vdash E_1 : \mathit{intloc} \qquad \pi \vdash E_2 : \mathit{intexp}}{\pi \vdash E_1 := E_2 : \mathit{comm}} \qquad \frac{\pi \vdash E_1 : \mathit{comm} \qquad \pi \vdash E_2 : \mathit{comm}}{\pi \vdash E_1 ; E_2 : \mathit{comm}}$$

$$\frac{\pi \vdash E_1 : \mathit{boolexp} \qquad \pi \vdash E_2 : \mathit{comm} \qquad \pi \vdash E_3 : \mathit{comm}}{\pi \vdash \textbf{if}\ E_1\ \textbf{then}\ E_2\ \textbf{else}\ E_3\ \textbf{fi} : \mathit{comm}} \qquad \pi \vdash \textbf{skip} : \mathit{comm} \qquad \frac{\vdash E_1 : \mathit{boolexp} \qquad \vdash E_2 : \mathit{comm}}{\vdash \textbf{while}\ E_1\ \textbf{do}\ E_2\ \textbf{od} : \mathit{comm}}$$

$$\frac{\pi \vdash E_1 : \mathit{intexp} \qquad \pi \vdash E_2 : \mathit{intexp}}{\pi \vdash E_1 + E_2 : \mathit{boolexp}} \qquad \frac{\pi \vdash E_1 : \tau\ \mathit{exp} \qquad \pi \vdash E_2 : \tau\ \mathit{exp}}{\pi \vdash E_1 = E_2 : \mathit{intexp}} \qquad \frac{\pi \vdash E : \mathit{boolexp}}{\pi \vdash \neg E : \mathit{boolexp}}$$

$$\frac{\pi \vdash E : \mathit{intloc}}{\pi \vdash @E : \mathit{intexp}} \qquad \pi \vdash N : \mathit{intexp} \qquad \pi \vdash L : \mathit{intloc}$$

Higher-Order Programming Languages

- The imperative language with records and lambda abstraction:

$$\frac{\pi \vdash E : \Theta}{\pi \vdash I = E : \{I : \Theta\}} \qquad \frac{\pi \vdash E_1 : \pi_1 \quad \pi \vdash E_2 : \pi_2}{\pi \vdash E_1, E_2 : \pi_1 \dot{\cup} \pi_2} \qquad \frac{\pi \vdash E_1 : \pi_1 \quad \pi \overline{\cup} \pi_1 \vdash E_2 : \Theta}{\pi \vdash \mathit{with}\ E_1\ \mathit{do}\ E_2 : \Theta}$$

$$\pi \vdash I : \Theta \quad \text{if} \quad (I : \Theta) \in \pi \qquad \frac{\pi \overline{\cup} \{I : \Theta_1\} \vdash E : \Theta_2}{\pi \vdash \lambda I : \Theta_1 . E : \Theta_1 \to \Theta_2} \qquad \frac{\pi \vdash E_1 : \Theta_1 \to \Theta_2 \quad \pi \vdash E_2 : \Theta_1}{\pi \vdash E_1 E_2 : \Theta_2}$$

Higher-Order Programming Languages

- It was claimed that the abstraction principle is a record introduction principle → but it was limited how records were used
  - the records were named and referenced, for example:

    alias $A = loc_1$; **module** $M = \{$**fun** $F = @A + 1$, **proc** $P = A := F\}$ **in call** $M.P$

    but they were not written "in-line", as values in their own right: **alias** $A = loc_1$ **in call** $\{$**fun** $F = @A + 1$, **proc** $P = A := F\}.P$
- It was claimed that the parameterization principle was a lambda abstraction principle
  - the abstractions were parameterized, for example:

    fun $F(X : intexp) = X + 1$ **in** $A := F(@A) + 2$

    but arbitrary phrases were not parameterized:

    $A := ((\lambda\ X : intexp.X + 1)@A) + 2$

Higher-Order Programming Languages

- A language is *higher order* if it lets lambda abstractions (and records) be full-fledged values, to be used in line, as components of records and other structures, and as arguments and results of abstraction invocations.
- A language that does not allow this free use of lambda abstractions is *first order*.

• Higher order programming languages encourage computation on structures - lambda abstractions and records - rather than on just primitive values like integers.

The Semantics of Records and Lambda Abstractions

- The semantics of the core part of the language:

$$[[\pi \vdash L := E : comm]]e\,s = update([[\pi \vdash L : intloc]]e, [[\pi \vdash E : intexp]]e\,s, s)$$

$$[[\pi \vdash E_1; E_2 : comm]]e\,s = [[\pi \vdash E_2 : comm]]e([[\pi \vdash E_1 : comm]]e\,s)$$

…

$$[[\pi \vdash E_1 + E_2 : intexp]]e\,s = plus([[\pi \vdash E_1 : intexp]]e\,s, [[\pi \vdash E_2 : intexp]]e\,s)$$

…

$$[[\pi \vdash N : intexp]]e\,s = [[N : int]]e$$

$$[[\pi \vdash loc_i : intloc]]e = loc_i$$

where $[[comm]] = Store \to Store_\perp$ and $[[\tau\ exp]] = Store \to [[\tau]]$

Lazy Evaluation Semantics

- The semantics of lazily evaluated lambda abstractions and records:

$$[[\pi \vdash I = E : \{I : \Theta\}]]e = \{I = [[\pi \vdash E : \Theta]]e\}$$

$$[[\pi \vdash E_1, E_2 : \pi_1 \dot{\cup} \pi_2]]e = ([[\pi \vdash E_1 : \pi_1]]e) \cup ([[\pi \vdash E_2 : \pi_2]]e)$$

$$[[\pi \vdash \mathit{with}\ E_1\ \mathit{do}\ E_2 : \Theta]]e = [[\pi \overline{\cup} \pi_1 \vdash E_2 : \Theta]](e \overline{\cup} [[\pi \vdash E_1 : \pi_1]]e)$$

$$[[\pi \vdash I : \Theta]]e = v, \quad \text{where } (I = v) \in e$$

$$[[\pi \vdash \lambda I{:}\,\Theta_1 . E : \Theta_1 \to \Theta_2]]e = f, \quad \text{where } f\,u = [[\pi \overline{\cup} \{I : \Theta_1\} \vdash E : \Theta_2]](e \overline{\cup} \{I = u\})$$

and $e \overline{\cup} \{I = u\} = \{I = u\} \cup (e - \{(I = v) \mid (I = v) \in e\})$

$$[[\pi \vdash E_1 E_2 : \Theta_2]]e = ([[\pi \vdash E_1 : \Theta_1 \to \Theta_2]]e)([[\pi \vdash E_2 : \Theta_1]]e)$$

where $[[\{i : \Theta_i\}_{i \in I}]] = \{i : [[\Theta_i]]\}_{i \in I}$ and $[[\Theta_1 \to \Theta_2]] = [[\Theta_1]] \to [[\Theta_2]]$

Lazy Evaluation Semantics

- Makes the typing rules sound
- Makes the copy rules for records and lambda abstractions sound
- The rules have the following form:

  with $(i = E_i)_{i \in I}$ *do* $E \Rightarrow [E_i / i]_{i \in I} E$, where $I \subseteq$ Identifier is a finite set

  $(\lambda I : \theta . E_1) E_2 \Rightarrow [E_2/I] E_2$

  - Here, $(i=E_i)_{i \in I}$ abbreviates the record $(i_1=E_1, i_2=E_2, \ldots, i_n=E_n)$, for $I=\{i_1, i_2, \ldots, i_n\}$.
  - $[E_i / i]_{i \in I}$ is a similar abbreviation for a simultaneous substitution.

Lazy Evaluation Semantics

- The summarized definition of substitution:

$[E/I](E_1 := E_2) = [E/I]E_1 := [E/I]E_2$

$[E/I](E_1;E_2) = [E/I]E_1;[E/I]E_2$

…

$[E/I]\ loc_i = loc_i$

$[E/I](J = E_1) = (J = [E/I]E_1)$

$[E/I](E_1,E_2) = [E/I]E_1,[E/I]E_2$

$[E/I](\mathit{with}\ E_1\ \mathit{do}\ E_2) = \mathit{with}\ [E/I]E_1\ \mathit{do}\ E_2$, if $\pi \vdash E_1 : \pi_1$ holds and $(I:\theta) \in \pi_1$

$[E/I](\mathit{with}\ E_1\ \mathit{do}\ E_2) = \mathit{with}\ [E/I]E_1\ \mathit{do}\ [E/I]E_2$, if $\pi \vdash E_1 : \pi_1$ holds and $(I:\theta) \notin \pi_1$

$[E/I]I = E$

$[E/I]J = J$, and $J \neq I$

$[E/I](\lambda I:\theta.E_1) = \lambda I:\theta.E_1$

$[E/I](\lambda J:\theta.E_1) = \lambda J:\theta.[E/I]E_1$, if $I \neq J$ and $J \notin \mathit{FV}(E)$

$[E/I](\lambda J:\theta.E_1) = \lambda K:\theta.[E/I][K/J]E_1$, if $I \neq J$, $J \in \mathit{FV}(E)$, and K is fresh

$[E/I](E_1E_2) = [E/I]E_1[E/I]E_2$

Lazy Evaluation Semantics

- Adding variable declarations to the language

  - preserving the elegant lazy evaluation semantics by using a new block construction specifically for integer variables:

$$\frac{\pi \cup \{V : intloc\} \vdash E : comm}{\pi \vdash new\ V\ in\ E : comm}$$

- The block *new* V E → allocates a location, binds it to V, and lets V be visible to command E
- The denotational semantics:

$$[[\pi \vdash new\ V\ in\ E : comm]]\, e\, s = free\,(size\text{-}of\ s)\, s_2$$

where $(l, s_1) = allocate\ s$

and $s_2 = [[\pi \cup \{V : intloc\} \vdash E : comm]](e \cup \{V = l\})\, s_1$

Eager Evaluation Semantics

- The semantics of eagerly evaluated lambda abstractions and records:

$$[[\pi \vdash \mathrm{I} = \mathrm{E} : \{\mathrm{I} : \theta\}]]es = \{\mathrm{I} = [[\pi \vdash \mathrm{E} : \theta]]es\}$$

$$[[\pi \vdash \mathrm{E}_1, \mathrm{E}_2 : \pi_1 \dot{\cup} \pi_2]]es = ([[\pi \vdash \mathrm{E}_1 : \pi_1]]es) \cup ([[\pi \vdash \mathrm{E}_2 : \pi_2]]es)$$

$$[[\pi \vdash \mathit{with}\ \mathrm{E}_1\ \mathit{do}\ \mathrm{E}_2 : \theta]]es = [[\pi \overline{\cup} \pi_1 \vdash \mathrm{E}_2 : \theta]](e \overline{\cup} [[\pi \vdash \mathrm{E}_1 : \pi_1]]es)s$$

$$[[\pi \vdash \mathrm{I} : \theta]]es = u, \quad \text{where } (\mathrm{I} = u) \in e$$

$$[[\pi \vdash \lambda \mathrm{I} : \theta_1 . \mathrm{E} : \theta_1 \to \theta_2]]es = f, \quad \text{where } f\,u\,s' = [[\pi \overline{\cup} \{\mathrm{I} : \theta_1\} \vdash \mathrm{E} : \theta_2]](e \overline{\cup} \{\mathrm{I} = u\})s'$$

$$[[\pi \vdash \mathrm{E}_1 \mathrm{E}_2 : \theta_2]]es = ([[\pi \vdash \mathrm{E}_1 : \theta_1 \to \theta_2]]e)([[\pi \vdash \mathrm{E}_2 : \theta_1]]es)s$$

where $[[\{i : \theta_i\}_{i \in I}]] = (\{i : \mathit{eval}(\theta_i)\}_{i \in I})_\perp$

$[[\theta_1 \to \theta_2]] = (\mathit{eval}(\theta_1) \to [[\theta_2]])_\perp$

and $\mathit{eval}(\{i : \theta_i\}_{i \in I}) = \{i : \mathit{eval}(\theta_i)\}_{i \in I}$

$\mathit{eval}(\theta_1 \to \theta_2) = \mathit{eval}(\theta_1) \to [[\theta_2]]$

$[[\mathit{comm}]] = (\mathit{Store} \to \mathit{Store})_\perp$

$[[\tau\ \mathit{exp}]] = (\mathit{Store} \to [[\tau]])_\perp$

$\mathit{eval}(\mathit{comm}) = \mathit{Store}$

$\mathit{eval}(\tau\ \mathit{exp}) = [[\tau]]$

Eager Evaluation Semantics

- Uses the store argument to force the evaluation of the phrases that bind to identifiers
- The soundness of the typing rules are preserved
- But, the role of the store in the semantics invalidates the copy rules

- Recall:
  - requires that the semantic binding operation is strict: $\{I = \bot\} = \bot$
  - requires that the function application is strict: $f(\bot) = \bot$
  - $\cup, \overline{\cup}$ strict: $\bot \overline{\cup} e = \bot \cup e = \bot$
  - $[[\pi \vdash E : \theta]] \bot s = \bot$

Lazy and Eager Evaluation Combined

- Integrating lazy and eager evaluation along with the **newint** construction into the same language

- The "roadmap" to the language → the structure of the type attributes:

$\Theta ::= \tau \mid \tau\ exp$

$\tau ::= int \mid bool \mid intloc \mid store \mid \pi \mid \Theta_1 \rightarrow \Theta_2$

$\pi ::= \{i : \Theta_i\}_{i \in I}$

Lazy and Eager Evaluation Combined

- A type attribute $\tau \rightarrow$ the type of an evaluated phrase
- $\tau exp \rightarrow$ the type of an unevaluated phrase that produces a τ-typed value when evaluated
- commands (unevaluated phrases)$\rightarrow$ have the type *store exp*
- The type structure phrase *newint* (unevaluated) $\rightarrow$ has type *intloc exp*
- numeral (evaluated) $\rightarrow$ has type *int*
- declaration **var** *X*:**newint** (evaluated) $\rightarrow$ has type {*X*:*intloc*}

- Important equivalences: $comm \equiv store\ exp$

$\delta\ class \equiv \delta\ exp$

$\pi\ dec \equiv \pi\ exp$

Lazy and Eager Evaluation Combined

- The syntax of the language consists of the core language, **newint**, and the following extensions:

$$E ::= E_1 := E_2 \mid E_1; E_2 \mid \ldots \mid \textbf{newint} \mid \textit{lazy}\ I = E \mid \textit{eager}\ I = E$$

$$\mid E_1, E_2 \mid \textit{with}\ E_1\ \textit{do}\ E_2 \mid I \mid \lambda I : \theta . E \mid E_1 E_2$$

Lazy and Eager Evaluation Combined

- Examples:

(i) **var** *X*:**newint** is coded *eager X* = **newint**

(ii) **class** *K*= **record** **var** *Y*:**newint, proc** *P* = **skip end** is coded

lazy K=(*eager Y*= **newint,** *lazy P*= **skip**)

var *R*:*K* is coded *eager R*=*K*

(iii) **module** *M*={**var** *X*:**newin**t, **proc** *P*= **skip**} is coded

eager M=(*eager Y* =**newint**, *lazy P*=**skip**)

Lazy and Eager Evaluation Combined

- The lambda abstraction is two constructions:

  λI:τ*exp*.E $\rightarrow$ accepts an unevaluated phrase as its argument

  λI:τ.E $\rightarrow$ demands an evaluated phrase for its argument

Lazy and Eager Evaluation Combined

- The interaction of unevaluated, evaluated, and side-effecting phrases makes assembly of a sound set of typing rules a fascinating exercise.

  - Few possibilities:

$\pi \vdash \textbf{newint} : \mathit{intloc}\ \mathit{exp}$ $\qquad \dfrac{\pi \vdash \mathrm{E} : \theta}{\pi \vdash \mathit{lazy}\ \mathrm{I} = \mathrm{E} : \{\mathrm{I} : \theta\}\ \mathit{exp}}$ $\qquad \dfrac{\pi \vdash \mathrm{E} : \tau\ \mathit{exp}}{\pi \vdash \mathit{eager}\ \mathrm{I} = \mathrm{E} : \{\mathrm{I} : \tau\}\ \mathit{exp}}$

  - The evaluation of *with* E_1 *do* E_2 requires that E_1 be evaluated to a record before E_2 proceeds. This suggests (at least) these two rules:

$$\frac{\pi \vdash \mathrm{E}_1 : \pi_1 \quad \pi \overline{\cup}\ \pi_1 \vdash \mathrm{E}_2 : \theta}{\pi \vdash \mathit{with}\ \mathrm{E}_1\ \mathit{do}\ \mathrm{E}_2 : \theta}$$

$$\frac{\pi \vdash \mathrm{E}_1 : \pi_1\ \mathit{exp} \quad \pi \overline{\cup}\ \pi_1 \vdash \mathrm{E}_2 : \tau\ \mathit{exp}}{\pi \vdash \mathit{with}\ \mathrm{E}_1\ \mathit{do}\ \mathrm{E}_2 : \tau\ \mathit{exp}} \quad \text{where } \tau \in \{\pi_2, \mathit{store}\}$$

Lazy and Eager Evaluation Combined

- The semantics:

$[[\pi \vdash \textbf{newint}: intloc\ exp]]es = allocate\ s$

$[[\pi \vdash lazy\ I = E : \{I : \theta\}\ exp]]es = (\{I = [[\pi \vdash E : \theta]]e\}, s)$

$[[\pi \vdash eager\ I = E : \{I : \tau\}\ exp]]es = (\{I = v\}, s'),\ \text{where}\ (v, s') = [[\pi \vdash E : \tau\ exp]]es$

$[[\pi \vdash I : \tau\ exp]]es = v\ s,\ \text{where}\ (I = v) \in e$

$[[\pi \vdash I : \tau]]es = v,\ \ \text{where}(I = v) \in e$

$[[\pi \vdash with\ E_1\ do\ E_2 : \theta]]es = [[\pi \cup \overline{\pi_1} \vdash E_2 : \theta]](e \cup \overline{[[\pi \vdash E_1 : \pi_1]]e})s$

$[[\pi \vdash with\ E_1\ do\ E_2 : \pi_2\ exp]]es = [[\pi \cup \overline{\pi_1} \vdash E_2 : \pi_2\ exp]](e \cup \overline{e_1})s_1$

$\text{where}(e_1, s_1) = [[\pi \vdash E_1 : \pi_1\ exp]]es$

$[[\pi \vdash with\ E_1\ do\ E_2 : store\ exp]]es = free(size\text{-}of\ s)s_2$

$\text{where}\ (e_1, s_1) = [[\pi \vdash E_1 : \pi_1\ exp]]es$

$\text{and}\ s_2 = [[\pi \cup \overline{\pi_1} \vdash E_2 : store\ exp]](e \cup \overline{e_1})s_1$

$[[\pi \vdash E_1 E_2 : \theta_2]]es = ([[\pi \vdash E_1 : \theta_1 \to \theta_2]]e)([[\pi \vdash E_2 : \theta_1]]e)s$

$[[\pi \vdash E_1 E_2 : \theta]]es = ([[\pi \vdash E_1 : \tau \to \theta]]e)([[\pi \vdash E_2 : \tau\ exp]]es)s$

where $[[\tau\ exp]] = Store \to [[\tau]]$

$[[int]] = Int_\perp, [[bool]] = Bool_\perp$

$[[\theta_1 \to \theta_2]] = (proper[[\theta_1]] \to [[\theta_2]])_\perp$

$[[\{i : \theta_i\}_{i \in I}]] = (\{i : proper[[\theta_i]]\}_{i \in I})_\perp$

$[[intloc]] = Location_\perp, [[store]] = Store_\perp$

and $properD = D - \{\perp\}$

Lambda Abstraction Alone

- Extending a core language with naming and parameter devices, but do not caring about record structures, classes and modules

→ using lambda abstraction alone as the foundation

- The key idea is that a block of the form **begin define** I=E **in** E_2 **end** is "desugared" into $((\lambda I{:}\theta.E_2)E_1)$

→ parameter binding simulates definition binding

Lambda Abstraction Alone

- The "desugaring" technique (structured into 3 stages of translation):

(i) Every parameterized abstraction **define** $I(I_1 : \theta_1, I_2 : \theta_2, \ldots, I_n : \theta_n) = U$, $n \geq 0$,

is translated as : **define** $I = (\lambda I_1 : \theta_1.(\lambda I_2 : \theta_2.\cdots(\lambda I_n : \theta_n.U)\cdots)))$

→ The nested lambda abstraction binds the actual parameters to the formal parameters one at a time.

(ii) Every block **begin define** $I_1 = U_1$, **define** $I_2 = U_2, \cdots,$ **define** $I_n = U_n$ **in** W **end**, $n \geq 0$,

is translated as : $((\cdots(((\lambda I_1 : \theta_1.(\lambda I_2 : \theta_2.\cdots(\lambda I_n : \theta_n.W)\cdots))U_1)U_2)\cdots)U_n)$

→ The independently declared abstracts bind to their names one at a time.

Lambda Abstraction Alone

(iii) Every invocation **invoke** $I(V_1, V_2, \ldots, V_n)$, $n \geq 0$,

is translated as : $((\cdots((I\ V_1)V_2)\cdots)V_n)$

→ The actual parameter tuple is split into its individual components to match the format of the nested lambda abstraction denoted by I.

Orthogonality

- A new construction is *orthogonal* to a programming language if

(i) The semantics of the original constructs in the language remain unchanged by the addition of the new construction.

(ii) If the new construction uses component phrases from the original language, then the semantics of the new construction is uniformly defined with respect to the component phrases.

- The record construction → orthogonal construction

  (its addition does not affect the semantics of the existing language constructions)

The Model of the Programming Language

- Judging the appropriateness of records and lambda abstractions
- *Category Theory* → sets down precise criteria for judging whether or not a construction is the right one
- Using notions of *categorical product* and *categorical exponentiation*

The Model of the Programming Language

- Recall the notion of *cartesian product* and *function set* from set theory:
  - For sets A and B, *the cartesian product set*, $A \times B$, is the set of ordered pairs $\{\langle a, b\rangle \mid a \in A, b \in B\}$.

    Pairs are indexed by the operations $\downarrow 1$ and $\downarrow 2$, such that $\langle a, b\rangle \downarrow 1 = a$ and $\langle a, b\rangle \downarrow 2 = b$.

    Generalization from binary cartesian products, $A \times B$, to ternary products, $A \times B \times C$, and to n-ary products, $A_1 \times A_2 \times \ldots \times A_n$, for $n > 0$.
  - *A function set*, $A \Rightarrow B$, is the set of single-valued mappings from arguments in set A to answers in set B.

    Functions are "indexed" by *apply* operation:

    For $f \in A \Rightarrow B$, $apply\langle f, a\rangle = f(a)$.

The Model of the Programming Language

• When we add records to a core programming language, we are adding products, where a record type, $\{I_1{:}\theta_1, I_2{:}\theta_2, \ldots, I_n{:}\theta_n\}$,is an *n*-ary product.

→ Records seem to be ordered pairs, but does the semantics of records support this?

- **Definition1**:

  $\Theta_1 \times \Theta_2$ is the *categorical product* of Θ_1 and Θ_2 iff the following equalities hold:

  (i) $\langle E_1, E_2 \rangle \downarrow 1 \equiv E_1$

  (ii) $\langle E_1, E_2 \rangle \downarrow 2 \equiv E_2$

  (iii) $\langle E \downarrow 1, E \downarrow 2 \rangle \equiv E$, when $\pi \vdash E : \Theta_1 \times \Theta_2$ holds

→ $E_1 \equiv E_2$ is an abbreviation for $[[\pi \vdash E_1 : \Theta]]e = [[\pi \vdash E_2 : \Theta]]e$, where $e \in Env_\pi$

The Model of the Programming Language

- **Theorem1**: For the semantics of lazily evaluated lambda abstractions and records, $\Theta_1 \times \Theta_2 = \{fst : \Theta_1, snd : \Theta_2\}$ is the categorical product.

→ Fails for the eager evaluation semantics.

→ Although the record construction in eager evaluation semantics is not the categorical product, it is still valuable beause we can build tuples and index them most of the time → "tensor" product

The Model of the Programming Language

• When we add lambda abstractions to a programming language, we are adding a form of function set to the language, where

$\Theta_1 \Rightarrow \Theta_2$ is just $\Theta_1 \rightarrow \Theta_2$.

→ But does the semantics support the view that a lambda abstraction is a function?

- **Definition2**:

  $\Theta_1 \Rightarrow \Theta_2$ is the *categorical exponentiation* of Θ_1 and Θ_2 (with respect to a product $\Theta_1 \times \Theta_2$) iff the following equalities hold:

  (i) $apply\langle(\lambda I:\Theta_1.E),U\rangle \equiv subst(U,I,E)$, where $subst(U,I,E) \equiv with\ I = U\ do\ E$

  (ii) $(\lambda I:\Theta_1.apply\langle E,I\rangle) \equiv E$, if $\pi \vdash E:\Theta_1 \Rightarrow \Theta_2$ holds

The Model of the Programming Language

• Showing that the lazy evaluation semantics of lambda abstraction is appropriate:

- **Theorem2**: For the semantics of lazily evaluated lambda abstractions and records, $\Theta_1 \Rightarrow \Theta_2 = \Theta_1 \rightarrow \Theta_2$ is the categorical exponentiation.

→ Fails to hold for the eager evaluation semantics.

→ Clause(ii) of Definition2 fails, but Clause(i) does hold, and for this reason, we say that $\Theta_1 \rightarrow \Theta_2$ in the eager evaluation semantics is "weak exponentiation"

The Logic of the Programming Language

- Studying the typing rules and considering their logical properties

- Pretending that the set of type attributes for the core programming language are primitive propositions, as in logic
- Consider $\Theta_1 \wedge \Theta_2$: for it to be true, both Θ_1 and Θ_2 must be proved.

  Then, the two proofs are combined by an inference rule:

$$\frac{\pi \vdash \Theta_1 \qquad \pi \vdash \Theta_2}{\pi \vdash \Theta_1 \wedge \Theta_2}$$

$\pi \vdash \Theta$ → : "based upon assumptions, π, there is a proof of Θ"

The Logic of the Programming Language

- Inference rules for disassembling a conjuction proposition:

$$\frac{\pi \vdash \Theta_1 \wedge \Theta_2}{\pi \vdash \Theta_1} \qquad \frac{\pi \vdash \Theta_1 \wedge \Theta_2}{\pi \vdash \Theta_2}$$

- The typing rule for the assignment statment revised to read as follows:

$$\frac{\pi \vdash intloc \qquad \pi \vdash int\ exp}{\pi \vdash comm}$$

The Logic of the Programming Language

- Considering records – binary records of the form $\{fst : \Theta_1, snd : \Theta_2\}$.

  The typing rules follow:

$$\frac{\pi \vdash E_1 : \Theta_1 \qquad \pi \vdash E_2 : \Theta_2}{\pi \vdash \{fst = E_1, snd = E_2\} : \{fst : \Theta_1, snd : \Theta_2\}}$$

$$\frac{\pi \vdash E_1 : \{fst : \Theta_1, snd : \Theta_2\} \qquad \pi \cup \{fst : \Theta_1, snd : \Theta_2\} \vdash E_2 : \Theta}{\pi \vdash with\ E_1\ do\ E_2 : \Theta} \qquad \pi \vdash I : \Theta, \text{ if } (I : \Theta) \in \pi$$

- When the program components are omitted, we obtain inference rules for conjuction- $\{fst : \Theta_1, snd : \Theta_2\}$ is $\Theta_1 \wedge \Theta_2$:

$$\frac{\pi \vdash \Theta_1 \qquad \pi \vdash \Theta_2}{\pi \vdash \{fst : \Theta_1, snd : \Theta_2\}} \qquad \frac{\pi \vdash \{fst : \Theta_1, snd : \Theta_2\} \qquad \pi \cup \{\Theta_1, \Theta_2\} \vdash \Theta}{\pi \vdash \Theta}$$

$\pi \vdash \Theta$, if $(I : \Theta) \in \pi$

The Logic of the Programming Language

• The second and third rules can be combined to derive the traditional rules for conjunction

- instantiate Θ in the second rule by Θ_1 and then by Θ_2, respectively, and use the third rule both times to proe the second hypothesis of the second rule:

$$\frac{\pi \vdash \{fst : \Theta_1, snd : \Theta_2\}}{\pi \vdash \Theta_1} \qquad \frac{\pi \vdash \{fst : \Theta_1, snd : \Theta_2\}}{\pi \vdash \Theta_2}$$

The Logic of the Programming Language

• We can repeat the above procedure with the typing rules for lambda abstraction, and we obtain the usual rules for logical implication

- the type $\theta_1 \to \theta_2$ is the implication $\theta_1 \supset \theta_2$:

$$\frac{\pi \cup \{I : \theta_1\} \vdash \theta_2}{\pi \vdash \theta_1 \to \theta_2} \qquad \frac{\pi \vdash \theta_1 \to \theta_2 \quad \pi \vdash \theta_1}{\pi \vdash \theta_2} \qquad \pi \vdash \theta \text{ if } (I : \theta) \in \pi$$

The Logic of the Programming Language

- In this way, the typing rules for records and lambda abstractions define a propositional logic with conjuction and implication
- The inference rules that were derived for conjuction and implication are the traditional ones $\Rightarrow$ we have evidence that the typing rules for the programming language are appropriate

Thank you for your attention!