1 Introduction

In this lecture, we add constructs to the typed λ-calculus that allow working with more complicated data structures, such as pairs, tuples, records, sums and recursive functions. We also provide denotational semantics for these new constructs.

2 Recap – The Typed λ-Calculus $\lambda^{\rightarrow}$

2.1 Syntax

$$\begin{array}{lrcl} \text{terms} & e & ::= & n \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{null} \mid x \mid e_1\, e_2 \mid \lambda x{:}\tau.\, e \\ \text{types} & \tau & ::= & \mathbf{int} \mid \mathbf{bool} \mid \mathbf{unit} \mid \tau_1 \rightarrow \tau_2 \\ \text{values} & v & ::= & n \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{null} \mid \lambda x{:}\tau.\, e \text{ closed} \end{array}$$

2.2 Typing Rules

$$\Gamma \vdash n : \mathbf{int} \qquad \Gamma \vdash \mathbf{true} : \mathbf{bool} \qquad \Gamma \vdash \mathbf{false} : \mathbf{bool} \qquad \Gamma \vdash \mathbf{null} : \mathbf{unit} \qquad \Gamma, x{:}\tau \vdash x : \tau$$

$$\frac{\Gamma \vdash e_0 : \sigma \rightarrow \tau \quad \Gamma \vdash e_1 : \sigma}{\Gamma \vdash e_0\, e_1 : \tau} \qquad \frac{\Gamma, x{:}\sigma \vdash e : \tau}{\Gamma \vdash (\lambda x{:}\sigma.\, e) : \sigma \rightarrow \tau}$$

3 Simple Data Structures

Each data structure can be added by extending the syntax of expressions (e), types (τ) and values (v). The evaluation contexts (E) will also need to be extended, and evaluation and type derivation rules added to work with the new syntax.

3.1 Pairs

Syntax:

$$\begin{array}{rcl} e & ::= & \cdots \mid (e_1, e_2) \mid \mathbf{\#1}\, e \mid \mathbf{\#2}\, e \\ \tau & ::= & \cdots \mid \tau_1 * \tau_2 \\ v & ::= & \cdots \mid (v_1, v_2) \\ E & ::= & \cdots \mid (E, e) \mid (v, E) \mid \mathbf{\#1}\, E \mid \mathbf{\#2}\, E \end{array}$$

For every added syntactic form, we observe that we have expressions that *introduce* the form, and expressions that *eliminate* the form. In the case of pairs, the introduction expression is (e_1, e_2), and the elimination expressions are $\#1\, e$ and $\#2\, e$.

Evaluation rules:

$$\mathbf{\#1}\, (v_1, v_2) \longrightarrow v_1 \qquad \mathbf{\#2}\, (v_1, v_2) \longrightarrow v_2$$

Note that these rules define *eager* evaluation, because we only select from a pair when both elements are already evaluated to a value.

Typing rules:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash \mathbf{\#1}\, e : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash \mathbf{\#2}\, e : \tau_2}$$

3.2 Tuples

Syntax:

$$\begin{array}{rcl} e & ::= & \cdots \mid (e_1, \ldots, e_n) \mid \#\mathbf{n}\, e \\ \tau & ::= & \cdots \mid \tau_1 * \cdots * \tau_n \\ v & ::= & \cdots \mid (v_1, \ldots, v_n) \\ E & ::= & \cdots \mid (v_1, \ldots, v_{i-1}, E, e_{i+1}, \ldots, e_n) \mid \#\mathbf{n}\, E \end{array}$$

Evaluation rule:

$$\#m\,(v_1, \ldots, v_n) \longrightarrow v_m \qquad (1 \le m \le n)$$

Typing rules:

$$\frac{\Gamma \vdash e_i : \tau_i \quad i \in \{1, \ldots, n\}}{\Gamma \vdash (e_1, \ldots, e_n) : \tau_1 * \cdots * \tau_n} \qquad \frac{\Gamma \vdash e : \tau_1 * \cdots * \tau_n}{\Gamma \vdash \#m\, e : \tau_m} \qquad (1 \le m \le n)$$

3.3 Records

Syntax:

$$\begin{array}{rcl} e & ::= & \cdots \mid \{x_1 = e_1, \ldots, x_n = e_n\} \mid e.x \\ \tau & ::= & \cdots \mid \{x_1 : \tau_1, \ldots, x_n : \tau_n\} \\ v & ::= & \cdots \mid \{x_1 = v_1, \ldots, x_n = v_n\} \\ E & ::= & \cdots \mid \{x_1 = v_1, \ldots, x_{i-1} = v_{i-1}, x_i = E, x_{i+1} = e_{i+1}, \ldots, x_n = e_n) \mid E.x \end{array}$$

Evaluation rule:

$$\{x_1 = v_1, \ldots, x_n = v_n\}.x_i \longrightarrow v_i \qquad (1 \le i \le n)$$

Typing rules:

$$\frac{\Gamma \vdash e_i : \tau_i,\ 1 \le i \le n}{\Gamma \vdash \{x_1 = e_1, \ldots, x_n = e_n) : \{x_1 : \tau_1, \ldots, x_n : \tau_n\}} \qquad \frac{\Gamma \vdash e : \{x_1 : \tau_1, \ldots, x_n : \tau_n\}}{\Gamma \vdash e.x_i : \tau_i} \qquad (1 \le i \le n)$$

3.4 Sums

Sums are useful for representing datatypes that can have multiple forms. For example, a tail of a list can either be another nonempty list or **null**.

Syntax:

$$\begin{array}{rcl} e & ::= & \cdots \mid \mathbf{inl}_{\tau_1+\tau_2} e \mid \mathbf{inr}_{\tau_1+\tau_2} e \mid \mathbf{case}\, e_0\, \mathbf{of}\, x.e_1 \mid y.e_2 \\ \tau & ::= & \cdots \mid \tau_1 + \tau_2 \\ v & ::= & \cdots \mid \mathbf{inl}_{\tau_1+\tau_2}\, v \mid \mathbf{inr}_{\tau_1+\tau_2}\, v \\ E & ::= & \cdots \mid \mathbf{inl}\, E \mid \mathbf{inr}\, E \mid \mathbf{case}\, E\, \mathbf{of}\, x.e_1 \mid y.e_2 \end{array}$$

The **inl** and **inr** constructs are called *left injection* and *right injection*, respectively.

Evaluation rules:

$$\mathbf{case}\ (\mathbf{inl}_{\tau_1+\tau_2}\, v)\ \mathbf{of}\ x.e_1 \mid y.e_2 \longrightarrow e_1\{v/x\} \qquad \mathbf{case}\ (\mathbf{inr}_{\tau_1+\tau_2}\, v)\ \mathbf{of}\ x.e_1 \mid y.e_2 \longrightarrow e_2\{v/y\}$$

Here e_1 and e_2 are functions and must have the same codomain type in order for the whole **case** expression to have a type.

Typing rules:

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathbf{inl}_{\tau_1+\tau_2}\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathbf{inr}_{\tau_1+\tau_2}\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e_0 : \tau_1 + \tau_2 \quad \Gamma, x{:}\tau_1 \vdash e_1 : \tau_3 \quad \Gamma, y{:}\tau_2 \vdash e_2 : \tau_3}{\Gamma \vdash \mathbf{case}\ e_0\ \mathbf{of}\ x.e_1 \mid y.e_2 : \tau_3}$$

To give an example of the sum type, consider the sum of two **unit** types, $\mathbf{unit} + \mathbf{unit}$. This type has exactly two elements, namely **inl null** and **inr null**. We could take this as a definition of the type **bool** with elements $\mathbf{true} \triangleq \mathbf{inl\ null}$ and $\mathbf{false} \triangleq \mathbf{inr\ null}$. The statement **if** b **then** e_1 **else** e_2 could then be written as **case** b **of** $z.e_1 \mid z.e_2$.

SML and OCaml have *variant* or *algebraic datatypes*, which generalize sum types. (Variants in ML can also be recursive, which we will not model here.)

$$\begin{array}{rcl} e & ::= & \cdots \mid x_i(e) \\ \tau & ::= & \cdots \mid [x_1{:}\tau_1, \ldots, x_n{:}\tau_n] \end{array}$$

Recall that the ML syntax is

```
datatype t = x1 of t1 | ... | xn of tn.
```

The `xi` are constructors, and must be globally (across all types) unique to avoid confusion as to which type a particular constructor is referring to (in our sum type, the confusion is alleviated by using $\tau_1 + \tau_2$ subscripts in **inl** and **inr**).

4 Denotational Semantics

We now give the denotational semantics for type domains of $\lambda^{\rightarrow+*}$, the strongly-typed λ-calculus with sum and product types.

$$\begin{array}{rcl} \mathcal{T}[\![\tau \rightarrow \tau']\!] & = & \mathcal{T}[\![\tau]\!] \rightarrow \mathcal{T}[\![\tau']\!] \\ \mathcal{T}[\![\tau * \tau']\!] & = & \mathcal{T}[\![\tau]\!] \times \mathcal{T}[\![\tau']\!] \\ \mathcal{T}[\![\tau + \tau']\!] & = & \mathcal{T}[\![\tau]\!] + \mathcal{T}[\![\tau']\!] \end{array}$$

As before, our contract for this language is:

$$\rho \models \Gamma \ \wedge\ \Gamma \vdash e : \tau \quad \Rightarrow \quad \mathcal{C}[\![\Gamma \vdash e : \tau]\!]\rho \ \in\ \mathcal{T}[\![\tau]\!].$$

The remaining semantic rules are:

$$\begin{array}{rcl} \mathcal{C}[\![\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2]\!]\rho & = & \langle \mathcal{C}[\![\Gamma \vdash e_1 : \tau_1]\!]\rho, \mathcal{C}[\![\Gamma \vdash e_2 : \tau_2]\!]\rho \rangle \\ \mathcal{C}[\![\Gamma \vdash \#\mathbf{1}\ e : \tau_1]\!]\rho & = & \pi_1(\mathcal{C}[\![\Gamma \vdash e : \tau_1 * \tau_2]\!]\rho) \\ \mathcal{C}[\![\Gamma \vdash \#\mathbf{2}\ e : \tau_2]\!]\rho & = & \pi_2(\mathcal{C}[\![\Gamma \vdash e : \tau_1 * \tau_2]\!]\rho) \\ \mathcal{C}[\![\Gamma \vdash \mathbf{inl}_{\tau_1+\tau_2}\ e : \tau_1 + \tau_2]\!]\rho & = & \mathrm{in}_1(\mathcal{C}[\![\Gamma \vdash e : \tau_1]\!]\rho) \\ \mathcal{C}[\![\Gamma \vdash \mathbf{inr}_{\tau_1+\tau_2}\ e : \tau_1 + \tau_2]\!]\rho & = & \mathrm{in}_2(\mathcal{C}[\![\Gamma \vdash e : \tau_2]\!]\rho) \\ \mathcal{C}[\![\Gamma \vdash \mathbf{case}\ e_0\ \mathbf{of}\ x.e_1 \mid y.e_2 : \tau_3]\!]\rho & = & \mathrm{case}\ \mathcal{C}[\![\Gamma \vdash e_0 : \tau_1 + \tau_2]\!]\rho\ \mathrm{of} \\ & & \quad x_1.(\mathcal{C}[\![\Gamma, x{:}\tau_1 \vdash e_1 : \tau_3]\!]\rho[x \mapsto x_1] \ \mid\ x_2.(\mathcal{C}[\![\Gamma, y{:}\tau_2 \vdash e_2 : \tau_3]\!]\rho[y \mapsto x_2] \end{array}$$

where π_n is the (mathematical) projection operator that selects the nth element of a product, $\mathrm{in_n}$ is the injection operator that injects an element into a coproduct, and $[\cdot, \cdot] \in (D_1 \rightarrow D_3) * (D_2 \rightarrow D_3) \rightarrow (D_1 + D_2 \rightarrow D_3)$

5 Adding Recursion

So far this language is not Turing-complete, because there is no way to do unbounded recursion. This is true because there is no possibility of nontermination. The easiest way to add this capability is to support recursive functions.

To do this, we first extend the definition of an expression:

$$e \quad ::= \quad \cdots \mid \mathbf{rec}\, f : \tau \to \tau' . \lambda x : \tau . e$$

The new keyword **rec** defines a recursive function named f such that both x and f are in scope inside e. Intuitively, the meaning of $\mathbf{rec}\, f : \tau \to \tau' . \lambda x : \tau . e$ is the least fixpoint of the map $f \mapsto \lambda x : \tau . e$, where both f and $\lambda x : \tau . e$ are of type $\tau \to \tau'$.

For example, we would write the recursive function

$$f(x) \quad = \quad \mathbf{if}\, x > 0\, \mathbf{then}\, 1\, \mathbf{else}\, f(x+1)$$

as

$$\mathbf{rec}\, f : \mathbf{int} \to \mathbf{int} . \lambda x : \mathbf{int} . \mathbf{if}\, x > 0\, \mathbf{then}\, 1\, \mathbf{else}\, f(x+1).$$

The small-step operational semantics evaluation rule for **rec** simply unfolds it whenever it is needed:

$$\mathbf{rec}\, f : \tau \to \tau' . \lambda x : \tau . e \quad \to \quad \lambda x : \tau . e\{(\mathbf{rec}\, f : \tau \to \tau' . \lambda x : \tau . e)/f\}$$

and the typing rule for **rec** is

$$\frac{\Gamma,\, f : \tau \to \tau',\, x : \tau \vdash e : \tau}{\Gamma \vdash (\mathbf{rec}\, f : \tau \to \tau' . \lambda x : \tau . e) : \tau \to \tau'}.$$

The denotational semantics is defined in terms of the **fix** operator on domains:

$$\mathcal{C}[\![\mathbf{rec}\, f : \tau \to \tau' . \lambda x : \tau . e]\!]\rho \quad = \quad \text{fix}\, \lambda g \in \mathcal{T}[\![\tau \to \tau']\!] . \lambda v \in \mathcal{T}[\![\tau]\!] . \mathcal{C}[\![\Gamma, f : \tau \to \tau', x : \tau \vdash e : \tau']\!]\rho[x \mapsto v, f \mapsto g]$$

Of course, whenever we take a fixed point, we have to make sure that a fixed point exists. We know that the function satisfies continuity and monotonicity because we are writing in the metalanguage. However, for a fixed point to exist, $\mathcal{T}[\![\tau \to \tau']\!]$ must be a pointed CPO. But for this to be true, we have to make sure $\perp$ is in the codomain of the function:

$$\mathcal{T}[\![\tau \to \tau]\!] \quad = \quad \mathcal{T}[\![\tau]\!] \to \mathcal{T}[\![\tau']\!]_\perp,$$

This will make the semantics of application potentially produce $\perp$, so we also have to change our contract to account for the possibility of nontermination:

$$\rho \models \Gamma \ \wedge \ \Gamma \vdash e : \tau \quad \Rightarrow \quad \mathcal{C}[\![e]\!]\rho \in \mathcal{T}[\![\tau]\!]_\perp.$$

Finally, we have to lift our semantics to take nontermination into account. For example, we should change the denotation of a pair to:

$$\mathcal{C}[\![\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2]\!]\rho \quad = \quad \begin{cases} \lfloor \langle \mathcal{C}[\![e_1]\!]\rho, \mathcal{C}[\![e_2]\!]\rho \rangle \rfloor, & \text{if both } \mathcal{C}[\![e_1]\!]\rho \neq \perp \text{ and } \mathcal{C}[\![e_2]\!]\rho \neq \perp, \\ \perp, & \text{otherwise.} \end{cases}$$

We can write this more conveniently using our metalanguage let construct:

$$\begin{aligned} \mathcal{C}[\![\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2]\!]\rho \quad = \quad & \text{let } v_1 \in \mathcal{T}[\![\tau_1]\!] = \mathcal{C}[\![\Gamma \vdash e_1 : \tau_1]\!]\rho \text{ in} \\ & \text{let } v_2 \in \mathcal{T}[\![\tau_2]\!] = \mathcal{C}[\![\Gamma \vdash e_2 : \tau_2]\!]\rho \text{ in} \\ & \lfloor \langle v_1, v_2 \rangle \rfloor \end{aligned}$$

Recall that mathematical "let" is defined as:

$$\text{let } x \in D = e_1 \text{ in } e_2 \quad = \quad (\lambda x \in D . e_2)^* (e_1)$$