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

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

Previously, with the untyped λ-calculus, we encoded integers and Booleans as λ-terms. In $\lambda^{\rightarrow}$, we are taking them as primitive constructs.

2.2 Typing Rules

$$\Gamma \vdash n : \mathsf{int} \qquad \Gamma \vdash \mathsf{true} : \mathsf{bool} \qquad \Gamma \vdash \mathsf{false} : \mathsf{bool} \qquad \Gamma \vdash \mathsf{null} : \mathsf{unit}$$

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

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:

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

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:

$$\#1\,(v_1, v_2) \to v_1 \qquad \#2\,(v_1, v_2) \to v_2$$

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:

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

3.2 Tuples

Syntax:

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

Evaluation rules:

$$\#m\,(v_1, \ldots, v_n) \to v_m, \quad 1 \le m \le n$$

Typing rules:

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

3.3 Records

A *record* is like a tuple with names—or, if you prefer, like a struct in C. Each entry is *labeled* with a name from a countable set of labels, Lab.

Syntax:

$$\begin{array}{rcl} l & \in & \text{Lab} \\ e & ::= & \cdots \mid \{l_1 = e_1, \ldots, l_n = e_n\} \mid e.x \\ \tau & ::= & \cdots \mid \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \\ v & ::= & \cdots \mid \{l_1 = v_1, \ldots, l_n = v_n\} \\ E & ::= & \cdots \mid \{l_1 = v_1, \ldots, l_{i-1} = v_{i-1}, l_i = E, l_{i+1} = e_{i+1}, \ldots, l_n = e_n) \mid E.x \end{array}$$

The names of the fields l_i are included as part of the type.

Evaluation rule:

$$\{l_1 = v_1, \ldots, l_n = v_n\}.l_i \to v_i, \quad 1 \le i \le n$$

Typing rules:

$$\frac{\Gamma \vdash e_i : \tau_i,\ 1 \le i \le n}{\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n) : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}} \qquad \frac{\Gamma \vdash e : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}{\Gamma \vdash e.l_i : \tau_i},\ 1 \le i \le n$$

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:

$$\begin{array}{rcl}
e & ::= & \cdots \;\mid\; \mathsf{inL}_{\tau_1+\tau_2} e \;\mid\; \mathsf{inR}_{\tau_1+\tau_2} e \;\mid\; \mathsf{match}\ e_0\ \mathsf{with}\ e_1 \,|\, e_2 \\
\tau & ::= & \cdots \;\mid\; \tau_1 + \tau_2 \\
v & ::= & \cdots \;\mid\; \mathsf{inL}_{\tau_1+\tau_2}\ v \;\mid\; \mathsf{inR}_{\tau_1+\tau_2}\ v \\
E & ::= & \cdots \;\mid\; \mathsf{inL}_{\tau_1+\tau_2}\ E \;\mid\; \mathsf{inR}_{\tau_1+\tau_2}\ E \;\mid\; \mathsf{match}\ E\ \mathsf{with}\ e_1 \,|\, e_2
\end{array}$$

The inL and inR constructs are called *left injection* and *right injection*, respectively.

Evaluation rules:

$$\mathsf{match}\ (\mathsf{inL}_{\tau_1+\tau_2}\ v)\ \mathsf{with}\ e_1 \,|\, e_2 \;\to\; e_1\, v \qquad\qquad \mathsf{match}\ (\mathsf{inR}_{\tau_1+\tau_2}\ v)\ \mathsf{with}\ e_1 \,|\, e_2 \;\to\; e_2\, v$$

Here e_1 and e_2 are functions and must have the same codomain type in order for the whole match expression to have a type. This formulation allows us to have heterogeneous sums. Note also that the evaluation of e_1 and e_2 are lazy: only one of them will be evaluated, and only after the choice has been made.

Typing rules:

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{inL}_{\tau_1+\tau_2}\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{inR}_{\tau_1+\tau_2}\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e_0 : \tau_1 + \tau_2 \quad \Gamma \vdash e_1 : \tau_1 \to \tau_3 \quad \Gamma \vdash e_2 : \tau_2 \to \tau_3}{\Gamma \vdash \mathsf{match}\ e_0\ \mathsf{with}\ e_1 \,|\, e_2 : \tau_3}$$

To give an example of the sum type, consider the sum of two unit types, $\mathsf{unit} + \mathsf{unit}$. This type has exactly two elements, namely $\mathsf{inL\ null}$ and $\mathsf{inR\ null}$. We could take this as a definition of the type bool with elements $\mathsf{true} \triangleq \mathsf{inL\ null}$ and $\mathsf{false} \triangleq \mathsf{inR\ null}$. The lazy conditional statement $\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ could then be written as $\mathsf{match}\ b\ \mathsf{with}\ \lambda z.\, e_1 \,|\, \lambda z.\, e_2$.

OCaml has a construct that is a generalization of the sum type. The OCaml syntax is

```
type t = C1 of t1 | ... | Cn of tn | D1 | ... | Dm.
```

Such datatypes are also called *variants*. The `Ci` and `Di` are *constructors*, and must be globally (across all types) unique to avoid confusion as to which type a particular constructor refers to (in our sum type, the ambiguity is resolved by decorating inL and inR with subscripts $\tau_1 + \tau_2$).

4 Denotational Semantics

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

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

As before, our contract for this language is:

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

The remaining semantic rules are:

$$
\begin{array}{rcl}
\mathcal{C}[\![(e_1, e_2)]\!]\,\Gamma\,\rho & \triangleq & \langle \mathcal{C}[\![e_1]\!]\,\Gamma\,\rho,\, \mathcal{C}[\![e_2]\!]\,\Gamma\,\rho\rangle \\
\mathcal{C}[\![\#1\ e]\!]\,\Gamma\,\rho & \triangleq & \pi_1(\mathcal{C}[\![e]\!]\,\Gamma\,\rho) \\
\mathcal{C}[\![\#2\ e]\!]\,\Gamma\,\rho & \triangleq & \pi_2(\mathcal{C}[\![e]\!]\,\Gamma\,\rho) \\
\mathcal{C}[\![\mathsf{inL}_{\tau_1+\tau_2}\ e]\!]\,\Gamma\,\rho & \triangleq & \iota_1(\mathcal{C}[\![e]\!]\,\Gamma\,\rho) \\
\mathcal{C}[\![\mathsf{inR}_{\tau_1+\tau_2}\ e]\!]\,\Gamma\,\rho & \triangleq & \iota_2(\mathcal{C}[\![e]\!]\,\Gamma\,\rho) \\
\mathcal{C}[\![\mathsf{match}\ e_0\ \mathsf{with}\ e_1 \,|\, e_2]\!]\,\Gamma\,\rho & \triangleq & \begin{cases} (\mathcal{C}[\![e_1]\!]\,\Gamma\,\rho)\,v, & \text{if } \mathcal{C}[\![e_0]\!]\,\Gamma\,\rho = \iota_1\,v, \\ (\mathcal{C}[\![e_2]\!]\,\Gamma\,\rho)\,v, & \text{if } \mathcal{C}[\![e_0]\!]\,\Gamma\,\rho = \iota_2\,v \end{cases} \\
& = & \mathit{match}\ \mathcal{C}[\![e_0]\!]\,\Gamma\,\rho\ \mathit{with}\ \iota_1\,v \to (\mathcal{C}[\![e_1]\!]\,\Gamma\,\rho)\,v \,|\, \iota_2\,v \to (\mathcal{C}[\![e_2]\!]\,\Gamma\,\rho)\,v,
\end{array}
$$

where π_n is the (mathematical) projection operator that selects the nth element of a product and ι_n is the injection operator that injects an element into a coproduct. These meta-operations are also well-typed, but we omit the annotations:

$$
\pi_i : \mathcal{T}[\![\tau_1]\!] \times \mathcal{T}[\![\tau_2]\!] \ \to\ \mathcal{T}[\![\tau_i]\!] \qquad\qquad \iota_i : \mathcal{T}[\![\tau_i]\!] \ \to\ \mathcal{T}[\![\tau_1]\!] + \mathcal{T}[\![\tau_2]\!]
$$

for $i \in \{1, 2\}$.

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 to the language is to add support for recursive functions.

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

$$
e \quad ::= \quad \cdots \quad | \quad \mathsf{rec}\, f : \sigma \to \tau \,.\, \lambda x : \sigma \,.\, e
$$

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 $\mathsf{rec}\, f : \sigma \to \tau \,.\, \lambda x : \sigma \,.\, e$ is the least fixed point of the map $f \mapsto \lambda x : \sigma \,.\, e$, where both f and $\lambda x : \sigma \,.\, e$ are of type $\sigma \to \tau$.

For example, we would write the recursive function

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

as

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

The small-step operational semantics evaluation rule for rec is:

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

and the typing rule for rec is

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

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

$$
\mathcal{C}[\![\mathsf{rec}\, f : \sigma \to \tau \,.\, \lambda x : \sigma \,.\, e]\!]\,\Gamma\,\rho \quad \triangleq \quad \mathsf{fix}\ \lambda g \in \mathcal{T}[\![\sigma \to \tau]\!] \,.\, \lambda v \in \mathcal{T}[\![\sigma]\!] \,.\, \mathcal{C}[\![e]\!]\,\Gamma[(\sigma \to \tau)/f,\, \sigma/x]\,\rho[v/x,\, g/f]
$$

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, $\mathcal{T}[\![\sigma \to \tau]\!]$ must be a pointed CPO. But for this to be true, we have to make sure $\bot$ is in the codomain of the function. We therefore redefine

$$\mathcal{T}[\![\mathsf{int}]\!] \triangleq \mathbb{Z}_\bot \qquad \mathcal{T}[\![\mathsf{bool}]\!] \triangleq \mathbb{2}_\bot \qquad \mathcal{T}[\![\mathsf{unit}]\!] \triangleq \{\mathsf{null}\}_\bot$$

and define the functions and product domains $\mathcal{T}[\![\sigma \to \tau]\!]$ and $\mathcal{T}[\![\sigma * \tau]\!]$ inductively as before, which will now be pointed CPOs.

In the disjoint sum (coproduct) domain construction shown in class, we tagged the elements of the two domains and took the union. This resulted in a non-pointed CPO, even if the two domains were pointed CPOs, so we must add a new bottom element $\bot$. But this construction is quite unsatisfactory, since iterating it leads to the proliferation of useless bottoms. Instead, we use the following alternative construction. Given pointed CPOs D and E, form the domain consisting of the set

$$\{\iota_1(d) \mid d \in D \wedge d \neq \bot_D\} \;\cup\; \{\iota_2(e) \mid e \in E \wedge e \neq \bot_E\} \;\cup\; \{\bot\},$$

where $\bot$ is a new element, ordered by

$$x \sqsubseteq y \quad \overset{\triangle}{\iff} \quad x = \bot \;\vee\; (x = \iota_1(d) \wedge y = \iota_1(d') \wedge d \sqsubseteq_D d') \;\vee\; (x = \iota_2(e) \wedge y = \iota_2(e') \wedge e \sqsubseteq_E e').$$

This is called the *smash sum* of D and E.

We also have to change our contract to account for the possibility of nontermination:

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

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

$$\mathcal{C}[\![(e_1, e_2)]\!]\,\Gamma\,\rho \quad \triangleq \quad \begin{cases} \langle \mathcal{C}[\![e_1]\!]\,\Gamma\,\rho,\, \mathcal{C}[\![e_2]\!]\,\Gamma\,\rho\rangle, & \text{if both } \mathcal{C}[\![e_1]\!]\,\Gamma\,\rho \neq \bot \text{ and } \mathcal{C}[\![e_2]\!]\,\Gamma\,\rho \neq \bot, \\ \bot, & \text{otherwise.} \end{cases}$$

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

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

where $\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2$. Recall that mathematical *let* is defined as:

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