Scribe: Mehmet Fidanboylu, Matthew Schultz Lecturer: Radu Rugina

1 Previously...

Last time, we extended $\lambda^{\rightarrow}$ to $\lambda^{\rightarrow\times+}$. We added pairs and case construct to our language extending e to:

e:=... | $\langle e_0, e_1\rangle$|fst e|snd e|
inl e as $\tau_1 + \tau_2$|inr e as $\tau_1 + \tau_2$| case e_0 of $e_1|e_2$

Then we extended the types to include the product (pairs) and disjoint sums (case) of two types.

τ = int | bool | unit $|\tau_1 \rightarrow \tau_2|\tau_1 \times \tau_2|\tau_1 + \tau_2$

We also extended the value definition to include the values in the form of pairs and left/right injective functions:

V = $p|\lambda x : \tau \bullet e|\langle v_0, v_1\rangle$|inl v as$\tau_1 + \tau_2$|inr v as$\tau_1 + \tau_2$

Also recall that
$\mathcal{T}[\![\tau_1 + \tau_2]\!] = \mathcal{T}[\![\tau_1]\!] + \mathcal{T}[\![\tau_2]\!]$
$\mathcal{T}[\![\tau_1 \times \tau_2]\!] = \mathcal{T}[\![\tau_1]\!] \times \mathcal{T}[\![\tau_2]\!]$

2 Operational Semantics

We will just give the axioms here and leave the rest as an exercise. Those can be derived quite easily using the semantics we had before without types.

The simple β-reduction: $\overline{(\lambda x : \tau \bullet e)e' \rightarrow e[x \rightarrow e']}$

Pairs: $\overline{fst\langle v_0, v_1\rangle \rightarrow v_0}$ $\quad$ $\overline{snd\langle v_0, v_1\rangle \rightarrow v_1}$

Notice that since these are axioms, the pairs can only have values not expressions.

Case: $\overline{\text{case(inl v as } \tau_1 + \tau_2\text{) of } e_1|e_2 \rightarrow e_1 v}$ $\quad$ $\overline{\text{case(inr v as } \tau_1 + \tau_2\text{) of } e_1|e_2 \rightarrow e_2 v}$

3 Typing Rules for Case and Pairs

The typing rules for other constructs do not change, we will just add new typing rules to handle pairs and case construct.

For a pair, we want to check that in any typing context Γ, the first element of the pair has type τ_0 and the second has type τ_1.

$$\frac{\Gamma \vdash e_0 : \tau_0 \qquad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash \langle e_0, e_1\rangle : \tau_0 \times \tau_1}$$

$$\frac{\Gamma \vdash e : \tau_0 \times \tau_1}{\Gamma \vdash \text{fst e} : \tau_0}$$

Similarly for snd,

$$\frac{\Gamma \vdash e : \tau_0 \times \tau_1}{\Gamma \vdash \text{snd e} : \tau_1}$$

Enforce e to have a product type and match the fst with tau_0 and snd with tau_1

Injective functions select the left or right of the disjoint sum of domains that e has as its type. Therefore the typing rules for inl and inr are simply:

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \text{inl e as } \tau_1 + \tau_2 : \tau_1 + \tau_2}$$
$$\frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \text{inr e as } \tau_1 + \tau_2 : \tau_1 + \tau_2}$$

The typing rule for case construct is again straight-forward. If the resulting case construct is to have type τ then e_0 should be one of the injective functions and therefore have type $\tau_1 + \tau_2$, and e_1,e_2 should have arrow types (if you look at the operational semantics, you can see that e_1, e_2 are actually functions) which take a value in the corresponding domain selected by the injective function and return a value in the resulting domain τ. Therefore, the typing rule is:

$$\frac{\Gamma \vdash e_0 : \tau_1 + \tau_2 \quad \Gamma \vdash e_1 : \tau_1 \rightarrow \tau \quad \Gamma \vdash e_2 : \tau_2 \rightarrow \tau}{\Gamma \vdash \text{case } e_0 \text{ of } e_1 | e_2 : \tau}$$

3.1 Why type annotations?

Let's leave out annotations for a second and consider the case where we have:

$$\begin{array}{ll} \text{case (inl 5) of} & \lambda x : int \bullet x \\ & |\lambda b : bool \bullet 0 \\ \text{and} & \\ \text{case (inl 5) of} & \lambda x : int \bullet x \\ & |\lambda y : int \bullet 0 \end{array}$$

In the first case, the type is int+bool, in the second case we have int+int. Without type annotations the type of (inl 5) is not unique! Uniqueness of types is something we assumed as a property of simply typed lambda calculus. However, by using typing annotations, we are able to differentiate the two types protecting the uniqueness property of types.

inl 5 as int+bool
inl 5 as int+int

4 Denotational Semantics

In this part, we will give the denotational semantics for the newly added constructs. The semantics for abstractions and applications will be the same and will not be repeated here.

$$
\begin{array}{rcl}
\mathcal{C}[\![\langle e_0, e_1\rangle]\!]\rho & = & \langle \mathcal{C}[\![e_0]\!]\rho, \mathcal{C}[\![e_1]\!]\rho\rangle \\
\mathcal{C}[\![\text{fst e}]\!]\rho & = & \pi_1(\mathcal{C}[\![e]\!]\rho) \\
\mathcal{C}[\![\text{snd e}]\!]\rho & = & \pi_2(\mathcal{C}[\![e]\!]\rho) \\
\mathcal{C}[\![\text{inl e as } \tau_1 + \tau_2]\!]\rho & = & in_1(\mathcal{C}[\![e]\!]\rho) \text{ where } in_1 : \mathcal{T}[\![\tau_1]\!] \rightarrow \mathcal{T}[\![\tau_1 + \tau_2]\!] \\
\mathcal{C}[\![\text{inr e as } \tau_1 + \tau_2]\!]\rho & = & in_2(\mathcal{C}[\![e]\!]\rho) \text{ where } in_2 : \mathcal{T}[\![\tau_2]\!] \rightarrow \mathcal{T}[\![\tau_1 + \tau_2]\!] \\
\mathcal{C}[\![\text{case e of } e_1|e_2]\!]\rho & = & case(\mathcal{C}[\![e]\!]\rho) \text{ of } in_1(x_1) \Rightarrow (\mathcal{C}[\![e_1]\!]\rho)x_1 \\
 & & |in_2(x_2) \Rightarrow (\mathcal{C}[\![e_2]\!]\rho)x_2
\end{array}
$$

The denotational semantics clearly specify how case is executed. Say, we have some expression x_1 which is mapped by in_1 to the disjoint sum of types $\tau_1 + \tau_2$ (as defined by the denotations of inl and inr). Then, we are going to call e_1 on this expression. You can see that x_1 has type τ_1 which agrees with e_1's domain type (see typing rules for injective functions).

5 Soundness Reminder

Just a little reminder what soundess was again

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

Where $\rho \models \Gamma$ means that ρ agrees with Γ.

6 Recursive Functions

Of course the big question is what happens with recursive functions and typing rules. Recursive functions are an extension of our expressions given thus far.

$$e := ...|recy : \tau \rightarrow \tau' \equiv (\lambda x : \tau.e)$$

So we want to add typing constructs for the recursive construct.

6.1 Recursive Typing Rule

The recursive typing rule that we will use is below

$$\frac{\Gamma[x \rightarrow \tau][y \rightarrow (\tau \rightarrow \tau')] \vdash e : \tau'}{\Gamma \vdash recy : \tau \rightarrow \tau'.(\lambda x : \tau.e) : \tau \rightarrow \tau'}$$

Now let's examine what happens in the denotational semantics with this typing rule.

7 Denotational Semantics of Recursion

The new denotation of the recy construct with types is given again in a fixed point form.

$$\mathcal{C}[\![recy : \tau \to \tau'.(\lambda x : \tau.e)]\!]\rho = fix(\lambda f \in \mathcal{T}[\![\tau \to \tau']\!].\lambda v \in \mathcal{T}[\![\tau]\!].\mathcal{C}[\![e]\!]\rho[x \to v][y \to f]$$

We'll focus on the abstraction case

$$\mathcal{C}[\![\lambda x : \tau.e]\!]\rho = \lambda v \in \mathcal{T}[\![\tau]\!].\mathcal{C}[\![e]\!]\rho[x \to v]$$

This is very similar. We need a function environment or a higher order function that has domain $\mathcal{T}[\![\tau \to \tau']\!] \to \mathcal{T}[\![\tau \to \tau']\!]$. Recall that f:D $\to$ D, where D is a cpo with bottom elements then the fixed point is $\bigsqcup f^n(\bot)$.

So we need $\mathcal{T}[\![\tau \to \tau']\!]$ to include a bottom element in case it doesn't terminate. But what does it mean for $\mathcal{T}[\![\tau \to \tau']\!]$ to include a bottom element? It means that the output (τ' might be a bottom element).

$$\mathcal{T}[\![\tau \to \tau']\!]_\bot \rho = \mathcal{T}[\![\tau]\!] \to \mathcal{T}[\![\tau']\!]_\bot$$

So now this rec construct models expressions that don't terminate. This requires us to redefine the denotational semantics of the rec construct to include the possibility of non-termination.

$$\mathcal{C}[\![recy : \tau \to \tau'.(\lambda x : \tau.e)]\!]\rho = fix(\lambda f \in \mathcal{T}[\![\tau \to \tau']\!]_\bot.\lambda v \in \mathcal{T}[\![\tau]\!].\mathcal{C}[\![e]\!]\rho[x \to v][y \to f]$$

We also need to redefine all the other semantics to accept the bottom element as input.

$$\begin{aligned}\mathcal{C}[\![e_0 e_1]\!]\rho \quad = \quad & let\ f = \mathcal{C}[\![e_0]\!]\rho\ in \\ & \quad let\ v = \mathcal{C}[\![e_1]\!]\rho\ in\ f(v)\end{aligned}$$

8 Type Equivalence

In this section we'll discuss what it means for two types to be equivalent. It is a non-trivial problem. In our earlier lectures equivalence just mean that $\tau = \tau'$, but later that will not suffice. Formally when we write $\tau \equiv \tau'$, we mean that τ, τ' are equivalent.

There are two types of equivalence that we will work with.

- *Equivalence Relation:* This relation has a few rules that govern it.

  Symmetry: $\dfrac{}{\tau \equiv \tau}$

  Reflexivity: $\dfrac{\tau_1 \equiv \tau_2}{\tau_2 \equiv \tau_1}$

  Transitivity: $\dfrac{\tau_1 \equiv \tau_2 \wedge \tau_2 \equiv \tau_3}{\tau_1 \equiv \tau_3}$

- *Compatible Relation* This relation also has a few rules that govern it.

  $\times$: $\dfrac{\tau_0 \equiv \tau_0' \wedge \tau_1 \equiv \tau_1'}{\tau_0 \times \tau_1 \equiv \tau_0' \times \tau_1'}$

  $\to$: $\dfrac{\tau_0 \equiv \tau_0' \wedge \tau_1 \equiv \tau_1'}{\tau_0 \to \tau_1 \equiv \tau_0' \to \tau_1'}$

  $+$: $\dfrac{\tau_0 \equiv \tau_0' \wedge \tau_1 \equiv \tau_1'}{\tau_0 + \tau_1 \equiv \tau_0' + \tau_1'}$

We also need a typing rule to infer when two types are equivalent.

$$\frac{\Gamma \vdash e : \tau \quad \tau \equiv \tau'}{\Gamma \vdash e : \tau'}$$

There are some actual problems in programming languages with regards to structures. These two equivalences are name equivalence and structural equivalence.

Assume we are given two structs.

$$struct\ foo\ \{int\ x, int\ y\}s;$$
$$struct\ bar\ \{int\ x, int\ y\}t;$$

In name equivalence they won't be equivalent because one struct is called s and one struct is called t. When we have a command such as s=t then we'll receive a type error. It should be mentioned that this is just a problem for structs. Primitives such as int,float,... in C are handled nicely.

In ML let's assume that we are also given two records.

$$val\ s = \{x = 0, y = 0\};$$
$$val\ t = \{x = 0, y = 0\};$$

In ML which has structural equivalence then the command s=t proceeds without an exception.