Lecture 04.1: Algebraic data types and general recursion

1. Recap

Recall the definition of the *simply typed lambda calculus*, a small programming language, from the previous class:

Type τ ::=	`int`	integer
	$\tau_1 \to \tau_2$	function
Term t ::=	n	number
	$t_1 + t_2$	addition
	x	variable
	$\lambda\ (x : \tau)\ .\ t'$	function definition
	$t_1\ t_2$	function application

This is a language that has two main concepts: functions and numbers. We can make numbers, add them together, create functions, call them, and so on. The semantics of numbers are not remarkable—adding them together works exactly as you expect—the main formalizations of interest here are functions and variables.

Specifically, variables in the lambda calculus are like the variables we're used to in mathematical functions. They represent *placeholders*, which we at some point replace with a concrete value.[1] Then, we think about functions as basically terms with variables in them, which we can choose to "call" (replace the argument variable). This enables our language to capture code reuse—i.e. we can wrap up a piece of code in a function and call it multiple times.

2. Algebraic data types

While functions are a great abstraction for code, our language needs better abstractions for data. We want to be able to represent relations between data, specifically the "and" and "or" relations—either I have a piece of data that has A *and* B as components, or it has A *or* B as components. We

[1] This interpretation is at odds with the normal definition of a variable in most programming languages, where variables can have their values reassigned. Really, those kinds of objects aren't variables but could instead be called *assignables*, or mutable slots that also have a name.

call these products and sums, respectively, and represent them in our grammar as follows:

Type τ ::=	$\ldots$	
	$\tau_1 \times \tau_2$	product
	$\tau_1 + \tau_2$	sum
Term t ::=	$\ldots$	
	(t_1, t_2)	pair
	$t.d$	projection
	$\texttt{inj}\ t = d\ \texttt{as}\ \tau$	injection
	$\texttt{case}\ t\ \{x_1 \hookrightarrow t_1 \mid x_2 \hookrightarrow t_2\}$	case
Direction d ::=	L	left
	R	right

2.1. Product types

A product type, or $\tau_1 \times \tau_2$, represents a term that has both an element of τ_1 and an element of τ_2. This is a simplified form of a similar kind of type found in other languages, like tuples in Python and structs in C. Pairs are the building block of composite data structures like classes. We can build pairs by taking two values and combining them, i.e. (t_1, t_2), and we can get our data out of a pair by using the *projection* operator, $t.L$ for the left element and $t.R$ for the right. For example, here's a function that adds the two elements of a pair:

$$\lambda\ (x : \texttt{int} \times \texttt{int})\ .\ x.L + x.R$$

Here's a function that takes a pair of a function and an int, and calls the function with the int:

$$\lambda\ (x : (\texttt{int} \to \texttt{int}) \times \texttt{int})\ .\ x.L\ x.R$$

We can generalize from pairs to n-tuples, or groups of n elements, by chaining together multiple pairs:

$$(1, (2, (3, 4))) : \texttt{int} \times (\texttt{int} \times (\texttt{int} \times \texttt{int}))$$

As always, we can formalize these language additions by providing statics and dynamics.

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (t_1, t_2) : \tau_1 \times \tau_2}\ \text{(T-pair)} \qquad \frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash t.L : \tau_1}\ \text{(T-project-L)} \qquad \frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash t.R : \tau_2}\ \text{(T-project-R)}$$

The first rule (T-pair) says that a pair (t_1, t_2) should have a product type corresponding to the types of its components, so if $t_1 : \tau_1$ and $t_2 : \tau_2$ then $(t_1, t_2) : \tau_1 \times \tau_2$. The next two rules define how to typecheck a projection, or accessing an element in a pair. The (T-project-L) says if we access the left element then we get the left type, and conversely the other rule says if we access the right element

then we get the right type. Next, we can define the dynamics:

$$\frac{t_1 \mapsto t_1'}{(t_1, t_2) \mapsto (t_1', t_2)}\ \text{(D-pair}_1\text{)} \qquad \frac{t_1\ \mathsf{val} \qquad t_2 \mapsto t_2'}{(t_1, t_2) \mapsto (t_1, t_2')}\ \text{(D-pair}_2\text{)} \qquad \frac{t_1\ \mathsf{val} \qquad t_2\ \mathsf{val}}{(t_1, t_2)\ \mathsf{val}}\ \text{(D-pair}_3\text{)}$$

$$\frac{t \mapsto t'}{t.d \mapsto t'.d}\ \text{(D-project}_1\text{)} \qquad \frac{(t_1, t_2)\ \mathsf{val}}{(t_1, t_2).L \mapsto t_1}\ \text{(D-project}_2\text{)} \qquad \frac{(t_1, t_2)\ \mathsf{val}}{(t_1, t_2).R \mapsto t_2}\ \text{(D-project}_3\text{)}$$

The first three rules say: to take a pair to value, we evaluate both of its components to a value. The next three rules say: to evaluate a project, we evaluate the projected term to a value (a pair), and then select the appropriate term out of the pair based on the requested direction.

Now we've defined a formal semantics for product types. We can create pairs and project elements out of them as well as typecheck terms involving pairs and projections.

2.2. Sum types

If product types define a term of two components A and B, a sum type represents a term of two possible components A or B, but not both. Sum types are an essential complement to product types and have existed in functional languages for decades, yet they still elude most mainstream programming languages.[2] For example, sums can be used to represent computations that either succeed or fail—this is like the option and result types we have seen. Sums are also used to represent recursive data structures like lists and syntax trees.

Sums are created by *injecting* a term into a sum, and they are eliminated by using a case statement with a branch for each form of the sum. For example, we can make a sum type like this:

$$(\texttt{inj}\ 1 = L\ \texttt{as}\ \texttt{int} + (\texttt{int} \to \texttt{int})) : \texttt{int} + (\texttt{int} \to \texttt{int})$$

And we can use that value like this:

$$\texttt{case}\ (\texttt{inj}\ 1 = L\ \texttt{as}\ \texttt{int} + (\texttt{int} \to \texttt{int}))\ \{x_1 \hookrightarrow x_1 + 1 \mid x_2 \hookrightarrow x_2\ 2\} \mapsto 1 + 1$$

As before, we can now define generalized statics and dynamics for sums.

$$\frac{\Gamma \vdash t : \tau_1}{\Gamma \vdash \texttt{inj}\ t = L\ \texttt{as}\ \tau_1 + \tau_2 : \tau_1 + \tau_2}\ \text{(T-inject-L)} \qquad \frac{\Gamma \vdash t : \tau_2}{\Gamma \vdash \texttt{inj}\ t = R\ \texttt{as}\ \tau_1 + \tau_2 : \tau_1 + \tau_2}\ \text{(T-inject-R)}$$

$$\frac{\Gamma \vdash t : \tau_1 + \tau_2 \qquad \Gamma, x_1 : \tau_1 \vdash t_1 : \tau \qquad \Gamma, x_2 : \tau_2 \vdash t_2 : \tau}{\Gamma \vdash \texttt{case}\ t\ \{x_1 \hookrightarrow t_1 \mid x_2 \hookrightarrow t_2\} : \tau}\ \text{(T-case)}$$

Our first two rules define how to typecheck injections. They say: if you ask for a term to be injected into a sum of a particular type, we verify that the injected term has the expected type. So if you inject $1 = L$ into $\texttt{int} + (\texttt{int} \to \texttt{int})$ then we verify that $1 : \texttt{int}$. The third rule defines the typing rule for cases. Cases are only valid on sum types (unlike OCaml match statements, which can

[2] In object-oriented programming languages, the closest thing to sum types is *subtyping* with classes, e.g. $B + C$ would be two subclasses B and C of some class A representing the sum type.

pattern match on any term like an integer), expressed by $t : \tau_1 + \tau_2$. Then we typecheck the two branches of the case to ensure that they return the same type. Lastly, the dynamics:

$$\frac{t \mapsto t'}{\texttt{inj}\ t = d\ \texttt{as}\ \tau \mapsto \texttt{inj}\ t' = d\ \texttt{as}\ \tau}\ (\text{D-inject}_1) \qquad \frac{t\ \mathsf{val}}{\texttt{inj}\ t = d\ \texttt{as}\ \tau\ \mathsf{val}}\ (\text{D-inject}_2)$$

$$\frac{t \mapsto t'}{\texttt{case}\ t\ \{x_1 \hookrightarrow t_1 \mid x_2 \hookrightarrow t_2\} \mapsto \texttt{case}\ t'\ \{x_1 \hookrightarrow t_1 \mid x_2 \hookrightarrow t_2\}}\ (\text{D-case}_1)$$

$$\frac{t\ \mathsf{val}}{\texttt{case inj}\ t = L\ \texttt{as}\ \tau\ \{x_1 \hookrightarrow t_1 \mid x_2 \hookrightarrow t_2\} \mapsto [x_1 \to t]\ t_1}\ (\text{D-case}_2)$$

$$\frac{t\ \mathsf{val}}{\texttt{case inj}\ t = R\ \texttt{as}\ \tau\ \{x_1 \hookrightarrow t_1 \mid x_2 \hookrightarrow t_2\} \mapsto [x_2 \to t]\ t_2}\ (\text{D-case}_3)$$

An injection is a value if the injected term is a value (otherwise we step it). The more interesting dynamic is for case: we step the argument to the case until we get the sum. The sum contains a direction which tells us which branch to execute, so we have a rule for each direction. Executing a branch is similar to calling a function–it just means substituting in a value (the injected term) for the variable corresponding to the given branch.

2.3. Type algebra

Collectively, these are called algebraic data types because they have algebraic properties similar to normal integers. Generally, you can understand these properties in terms of the *number of terms* that inhabit a particular type. For example, the type `bool` is inhabited by two terms, `true` and `false`. We write this as $|\texttt{bool}| = 2$.

To understand the algebraic properties of these types, we first need to add two concepts to our language:

$$\begin{array}{lll} \mathsf{Type}\ \tau ::= & \ldots & \\ & 0 & \text{void} \\ & 1 & \text{unit} \\ \\ \mathsf{Term}\ t ::= & \ldots & \\ & () & \text{unit} \end{array}$$

We introduce the types 0 and 1, sometimes called void and unit[3]. The idea behind these types is that there are 0 terms that have the type 0, and there is 1 term that has the type 1 (the unit term). With these in the languages, now we have an identity for both of our data types:

$$|\tau \times 1| = |\tau|$$
$$|\tau + 0| = |\tau|$$

[3]While the void type does not exist in OCaml, the unit type is used frequently to represent side effects.

For example, if $\tau = \texttt{bool}$, then the terms that have the type $\texttt{bool} \times 1$ are $(\texttt{true}, ())$ and $(\texttt{false}, ())$. In an information-theoretic sense, adding the unit type (or the 1 type) to our pair provides no additional information about our data structure. The number of terms inhabiting the type remain the same. Similarly, for the sum case, the type $\texttt{bool} + 0$ has two values: $\texttt{inj true} = L \texttt{ as bool} + 0$ and $\texttt{inj false} = L \texttt{ as bool} + 0$. There are no values of type 0, so there are no other possible injections.

More generally, the sum and product type operators follow these basic rules:

$$|\tau_1 \times \tau_2| = |\tau_1| \times |\tau_2|$$
$$|\tau_1 + \tau_2| = |\tau_1| + |\tau_2|$$

These rules then give rise to a number of algebraic properties. For example:

$$|\tau_1 \times \tau_2| = |\tau_2 \times \tau_1| \qquad \text{(commutativity)}$$
$$|(\tau_1 \times \tau_2) \times \tau_3| = |\tau_1 \times (\tau_2 \times \tau_3)| \qquad \text{(associativity)}$$
$$|\tau_1 \times (\tau_2 + \tau_3)| = |(\tau_1 \times \tau_2) + (\tau_1 \times \tau_3)| \qquad \text{(distributivity)}$$

This is a neat little observation, but if you want to do anything interesting with it, you have to dive a little deeper into the theory. There are still academic papers being published on novel ways to interpret the algebra of ADTs. For more on this, I recommend "The algebra (and calculus!) of algebraic data types" (uses Haskell syntax, but ideas are the same).

3. General recursion

One notable property of the simply typed lambda calculus (as we have defined it so far) is that every term *always* evaluates to a value (or an error).[4] We could, in fact, formally prove that is the case. Intuitively, the reason for this is that our language lacks *recursion*, or the ability for a term to reference itself. A consequence of this is that our language is no longer Turing-complete, i.e. it is not in the same complexity class of most general-purpose programming languages, so it cannot express as general a set of functions. Subsequently, adding recursion to our language is a desirable property not just from an ergonomics standpoint but also from the perspective of increasing the number of functions expressible in our language.[5]

When we write code, we often take function recursion for granted—it's just built in to the language semantics. For example, in Lua, we can write the factorial function like this:

```
local function fact(n)
  if n == 0 then return 1
  else return n * fact(n - 1) end
end
```

However, while we accept this works for functions, the same idea seems crazy for any other kind of term. We could not write something like this:

[4] In the literature, this property is called "strong normalization".

[5] If you are interested in exploring these types of problems more deeply (how can we represent a computation? How can we precisely understand the expressive power of a computation model?) then I would recommend taking CS 154.

```
local x = 0
local y = x + y
-- stdin:2: attempt to perform arithmetic on global 'y' (a nil value)
-- stack traceback:
--        stdin:2: in main chunk
--        [C]: in ?
```

Then the question becomes—what does a principled approach to recursion look like? First, let's develop a language extension that just enables recursion for functions. We won't concern ourselves with types right now, just the syntax and dynamics. We could create a new kind of function like:

$$\texttt{fn}\ f\ (x : \tau)\ .\ t$$

Where f is a variable that represents a handle to the function we're defining. For example, the factorial function would look like:

$$\texttt{fn}\ f\ (x : \texttt{int})\ .\ \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x * f(x-1)$$

We can define a dynamics for the general form that looks like this:

$$\frac{t_2\ \mathsf{val}}{(\texttt{fn}\ f\ (x : \tau)\ .\ t_1)\ t_2 \mapsto [x \to t_2, f \to (\texttt{fn}\ f\ (x : \tau)\ .\ t_1)]\ t_1}\ \text{(D-recapp)}$$

Intuitively, when we call a function, we replace any recursive references to the function with the function term itself. This is the core idea of *self-reference*—a term that refers to itself. For example, we can step through one layer of the factorial function:

$$\begin{aligned}
&(\texttt{fn}\ f\ (x : \texttt{int})\ .\ \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x * f(x-1))\ 2 \\
\mapsto\ &[x \to 2, f \to (\texttt{fn}\ f\ (x : \texttt{int})\ .\ \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x * f(x-1))]\ \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x * f(x-1) \\
=\ &\texttt{if}\ 2 = 0\ \texttt{then}\ 1\ \texttt{else}\ 2 * (\texttt{fn}\ f\ (x : \texttt{int})\ .\ \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x * f(x-1))(2-1) \\
\mapsto\ &\texttt{if}\ \texttt{false}\ \texttt{then}\ 1\ \texttt{else}\ 2 * (\texttt{fn}\ f\ (x : \texttt{int})\ .\ \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x * f(x-1))(2-1) \\
\mapsto\ &2 * (\texttt{fn}\ f\ (x : \texttt{int})\ .\ \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x * f(x-1))(2-1)
\end{aligned}$$

If we want to recursively call our function, we embed a copy of itself in itself. However, we tend to dislike special cases in programming languages, so we can generalize this mechanism to work for any term, not just functions. That is to say, we want to invent a language extension that permits an arbitrary term to get access to itself. We call this the *fixpoint* operator (as it has a strong relation to fixed-points in combinatory logic). It has the following form:

$$\texttt{fix}(\lambda\ (x : \tau)\ .\ t)$$

You can read this as "a term t where x = t". For example, if we want to compute an infinite sum, we can write that down as:

$$\begin{aligned}
&\texttt{fix}(\lambda\ (x : \texttt{int})\ .\ x + 1) \\
\mapsto\ &\texttt{fix}(\lambda\ (x : \texttt{int})\ .\ x + 1) + 1 \\
\mapsto\ &\texttt{fix}(\lambda\ (x : \texttt{int})\ .\ x + 1) + 1 + 1 \\
\mapsto\ &\ldots
\end{aligned}$$

We can provide it a formal semantics:

$$\frac{\Gamma \vdash t : \tau \to \tau}{\Gamma \vdash \texttt{fix}(t) : \tau} \text{ (T-fix)} \qquad \frac{t \mapsto t'}{\texttt{fix}(t) \mapsto \texttt{fix}(t')} \text{ (D-fix}_1\text{)}$$

$$\frac{}{\texttt{fix}(\lambda\ (x : \tau)\ .\ t) \mapsto [x \to \texttt{fix}(\lambda\ (x : \tau)\ .\ t)]\ t} \text{ (D-fix}_2\text{)}$$

The main rule to focus on is (D-fix$_2$). It has a similar form to our recursive function rule earlier, except instead of having two parameters (one for the recursive function and one for its parameter), now we just have a single variable representing the term being recursed. We can now rewrite the factorial function from before:

$$\texttt{fix}(\lambda\ (f : \texttt{int} \to \texttt{int})\ .\ \lambda\ (n : \texttt{int})\ .\ \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x * f(x - 1))$$

As an exercise for the reader, try calling this function with the number 3 and writing down the steps on a notepad until you reach a value (this function does happen to terminate, unlike the infinite sum). Here, the dynamics will be your guide.

With that, our language has generalized recursion. The key insight is that recursion comes from self-reference, and that self-reference must be exist as a language mechanism to enable recursion in typed languages.[6]

[6] In the *untyped* lambda calculus, you don't need explicit language support to create an equivalent fixpoint operator. That's why the untyped lambda calculus is actually Turing-complete, but you lose that expressiveness when you add a type system.