Recitation 4:
Products, Sums, and Generic Programming

15-312: Foundations of Programming Languages

Charles Yuan, Jeanne Luning Prak

February 7, 2018

Finite data structures in a programming language are created from the amalgamation of smaller structures, starting from the base types. Most useful structures can be constructed using two language features: **product** and **sum** types.

1 Products

The product of types τ_1 and τ_2, $\tau_1 \times \tau_2$, is the type of tuples (e_1, e_2) where $e_1 : \tau_1$ and $e_2 : \tau_2$. Products are familiar to functional programmers as a way of passing multiple arguments to functions and obtaining multiple results from them. They also represent the coupling together of several independent typed fields, since to have a value of product type, one must have a value in each of the product's fields.

Consider the following modification of System **T**, augmented with product types. Note the slightly different recursor, which now only binds one predecessor. We'll get back to that in a moment.

Typ	τ	$::=$	$\mathtt{nat}$	number
			$\tau_1 \to \tau_2$	function
			$\mathtt{unit}$	unit
			$\tau_1 \times \tau_2$	product
Exp	e	$::=$	x	variable
			$\mathtt{z}$	zero
			$\mathtt{s}(e)$	successor
			$\mathtt{iter}\{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x) \hookrightarrow e_1\}(e)$	recursion
			$\lambda\,(x : \tau)\,e$	abstraction
			$e_1(e_2)$	application
			$()$	empty pair
			(e_1, e_2)	pair
			$e \cdot \mathtt{l}$	left projection
			$e \cdot \mathtt{r}$	right projection

This is a language with binary products. The values of product type are created by the **pair** and **empty pair** constructors, their **introduction forms**, and they are converted back into their constituent types by the **left** and **right projections**, their **elimination forms**. Despite only

having binary products, we may encode n-ary products by nesting binary products in arbitrary order.

The `unit` type is the type of the empty product, with no fields. It has one value, the empty pair. While it may seem somewhat useless, seasoned ML programmers recognize it as the return type of functions with side-effects, the parameter type of suspended computations, etc. Though `unit` conveys no real data, it has tremendous utility in programming languages.

Mainstream programming languages often conflate `unit` with `void`, a different concept altogether, often speaking of functions of "`void` return type." In programming language theory we use the proper terminology, `unit`!

Examples of products:

$$\begin{gathered}
() : \mathtt{unit} \\
(\mathtt{z}, \mathtt{z}) : \mathtt{nat} \times \mathtt{nat} \\
(\mathtt{z}, (\mathtt{s}(\mathtt{z}), \mathtt{s}(\mathtt{z}))) : \mathtt{nat} \times (\mathtt{nat} \times \mathtt{nat}) \\
(\lambda\,(x : \mathtt{nat})\, x, \lambda\,(x : \mathtt{nat} \to \mathtt{nat})\, x) : (\mathtt{nat} \to \mathtt{nat}) \times ((\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat} \to \mathtt{nat})
\end{gathered}$$

Products are associative, so we often leave off the parentheses when the nesting order is arbitrary: $\mathtt{nat} \times \mathtt{nat} \times \mathtt{nat}$ instead of $\mathtt{nat} \times (\mathtt{nat} \times \mathtt{nat})$.

There are also alternative notations for product types: a tupled form

$$(\tau_1, \tau_2, \tau_3)$$

and various labeled forms

$$\langle \text{left} \hookrightarrow \tau_1, \text{right} \hookrightarrow \tau_2 \rangle$$

These notations typically mean exactly what they look like, and we use them to simplify our reasoning. In particular, labels are helpful to give names to the fields in a tuple.

The **projections** retrieve the left and right branches of a tuple in the natural way.

1.1 Statics

$$\frac{}{\Gamma \vdash () : \mathtt{unit}} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e \cdot \mathtt{l} : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e \cdot \mathtt{r} : \tau_2}$$

1.2 Dynamics

This is an eager dynamics for products. Think about how the rules would change for a lazy dynamics!

$$\frac{}{() \text{ val}} \qquad \frac{e_1 \text{ val} \quad e_2 \text{ val}}{(e_1, e_2) \text{ val}}$$

$$\frac{e_1 \longmapsto e_1'}{(e_1, e_2) \longmapsto (e_1', e_2)} \qquad \frac{e_2 \longmapsto e_2'}{(e_1, e_2) \longmapsto (e_1, e_2')}$$

$$\frac{e_1 \text{ val} \quad e_2 \text{ val}}{(e_1, e_2) \cdot \mathtt{l} \longmapsto e_1} \qquad \frac{e_1 \text{ val} \quad e_2 \text{ val}}{(e_1, e_2) \cdot \mathtt{r} \longmapsto e_2}$$

2 Sums

Sum types, denoted $\tau_1 + \tau_2$, are a tagged disjoint union of the types τ_1 and τ_2. That is, a value of type $\tau_1 + \tau_2$ contains either a value of type τ_1 or a value of type τ_2, along with the machinery to determine which "branch" is contained. The most common appearance of sum types is in ML, with algebraic datatypes:

```
datatype token = Number of int | Identifier of string | Semicolon
```

Here, a value of type `token` contains a number, an identifier, or a semicolon. Each possible branch contains a label and an internal type: `int`, `string`, or in the case of semicolon, `unit`.

We may represent the above type, without labels, as a sum:

$$\texttt{int} + \texttt{string} + \texttt{unit}$$

Note how sums are different from products: instead of containing one field of each of the types, it contains exactly one of the types.

Sums are *not* the same as the crude "unions" in the C family, in that a value of sum type stores which branch was taken, and attempting to view the value from any other branch is prohibited by the type system. They are *not* the same as enumerations in C, Java, Python, etc., which are largely incapable of storing internal data (and usually not typesafe either). And finally, they are also *not* the same as class hierarchies in so-called object-oriented languages, though classes are often used to emulate sum types to varying degrees of success.

Instead, sums are a typesafe manner of representing choice, giving the language flexibility without introducing unsafe type coercions or runtime checks.

We can extend System **T** with sums:

Typ	τ	::=	...	
			$\texttt{void}$	void
			$\tau_1 + \tau_2$	sum
Exp	e	::=	...	
			$\texttt{l}\{\tau_1;\tau_2\} \cdot e$	left injection
			$\texttt{r}\{\tau_1;\tau_2\} \cdot e$	right injection
			$\texttt{case}\ e\ \{\texttt{l} \cdot x_1 \hookrightarrow e_1 \mid \texttt{r} \cdot x_2 \hookrightarrow e_2\}$	case

This language contains binary sums, whose values are introduced by the **left** and **right injections**, and eliminated by the **case** expression.

It also contains the type `void`, which is the **empty sum**. A type with no branches can have no values, so `void` is not inhabited by any values—it is truly empty. This is why it does not make sense for a function to return `void`; since there are no values of this type, if anything it would mean that the function does not return at all!

The injections correspond to the two branches of a binary sum. An injection attaches a "label" to its operand, signifying that the result takes either the left or the right branch of the sum.

Since the type of the branch that was not taken is not given by e, we explicitly provide the types of both branches to the injections. When the types are clear, we may omit them in the syntax for a shorthand:

$$\texttt{l} \cdot e \qquad \texttt{r} \cdot e$$

The case expression decomposes a value of sum type, and depending on whether the contained value is of the left or right branch, binds it into either e_1 or e_2.

Examples of sums:

$$\mathtt{l} \cdot \mathtt{z} : \mathtt{nat} + \tau$$
$$\mathtt{r} \cdot \mathtt{s}(\mathtt{z}) : \tau + \mathtt{nat}$$
$$\mathtt{l} \cdot \mathtt{r} \cdot \lambda\,(x : \mathtt{nat})\, x : (\tau_1 + (\mathtt{nat} \to \mathtt{nat})) + \tau_2$$

Try to derive the types of these expressions, as it might not be obvious at first glance. We wrote that the types on the right contain the type τ, signifying that any type can be in that branch, depending on the type parameter we gave to the injection.

Now we look at the `case` expression:

$$\mathtt{case}\ \mathtt{l}\{\mathtt{nat} \to \mathtt{nat}; \mathtt{nat}\} \cdot \lambda\,(x : \mathtt{nat})\, x\ \{\mathtt{l} \cdot x_1 \hookrightarrow x_1(\mathtt{z}) \mid \mathtt{r} \cdot x_2 \hookrightarrow x_2\}$$

This `case` expression examines its operand, $\mathtt{l}\{\mathtt{nat} \to \mathtt{nat}; \mathtt{nat}\} \cdot \lambda\,(x : \mathtt{nat})\, x$, and in the left case binds its wrapped value to x_1 in $x_1(\mathtt{z})$. In the right case it would bind the wrapped value to x_2 in x_2.

There is a special variety of `case`, one with no branches, which works on values of type `void`:

$$\mathtt{case}\ e\ \{\}\ [\text{where } e : \mathtt{void}]$$

But wait! We just said there were no values of type `void`, so why do we even need this? Well, even though there are no such values, we can still write functions that take in arguments of type `void`:

$$\lambda\,(x : \mathtt{void})\, \mathtt{case}\ x\ \{\}$$

For us to be able to construct this function and have it satisfy type safety, we need some construct that eliminates the `void` type even if nothing introduces it.

Note: The current editions of PFPL have another construct, $\mathtt{abort}(e)$, which serves the same purpose as this empty `case` expression. According to the author of the textbook, that notation was a historical choice that mischaracterizes the construct, since it does not actually "abort" any computation.

One useful flavor of sum types is booleans:

$$\mathtt{bool} \triangleq \mathtt{unit} + \mathtt{unit}$$
$$\mathtt{true} \triangleq \mathtt{l} \cdot ()$$
$$\mathtt{false} \triangleq \mathtt{r} \cdot ()$$

2.1 Statics

$$\frac{\Gamma \vdash e : \mathtt{void}}{\Gamma \vdash \mathtt{case}\ e\ \{\} : \tau}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathtt{l}\{\tau_1; \tau_2\} \cdot e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathtt{r}\{\tau_1; \tau_2\} \cdot e : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathtt{case}\ e\ \{\mathtt{l} \cdot x_1 \hookrightarrow e_1 \mid \mathtt{r} \cdot x_2 \hookrightarrow e_2\} : \tau}$$

2.2 Dynamics

Like before, this is an eager dynamics, so think about how the rules would change for a lazy dynamics!

$$\frac{e \longmapsto e'}{\texttt{case}\ e\ \{\} \longmapsto \texttt{case}\ e'\ \{\}}$$

$$\frac{e\ \mathsf{val}}{\texttt{l} \cdot e\ \mathsf{val}} \qquad \frac{e\ \mathsf{val}}{\texttt{r} \cdot e\ \mathsf{val}}$$

$$\frac{e \longmapsto e'}{\texttt{l} \cdot e \longmapsto \texttt{l} \cdot e'} \qquad \frac{e \longmapsto e'}{\texttt{r} \cdot e \longmapsto \texttt{r} \cdot e'}$$

$$\frac{e \longmapsto e'}{\texttt{case}\ e\ \{\texttt{l} \cdot x_1 \hookrightarrow e_1 \mid \texttt{r} \cdot x_2 \hookrightarrow e_2\} \longmapsto \texttt{case}\ e'\ \{\texttt{l} \cdot x_1 \hookrightarrow e_1 \mid \texttt{r} \cdot x_2 \hookrightarrow e_2\}}$$

$$\frac{e\ \mathsf{val}}{\texttt{case}\ \texttt{l} \cdot e\ \{\texttt{l} \cdot x_1 \hookrightarrow e_1 \mid \texttt{r} \cdot x_2 \hookrightarrow e_2\} \longmapsto [e/x_1]e_1}$$

$$\frac{e\ \mathsf{val}}{\texttt{case}\ \texttt{r} \cdot e\ \{\texttt{l} \cdot x_1 \hookrightarrow e_1 \mid \texttt{r} \cdot x_2 \hookrightarrow e_2\} \longmapsto [e/x_2]e_2}$$

3 Recursor

When we were first introduced to System **T**, we questioned why it was necessary to have two binding sites in the primitive recursor. Now that we have product types, we can roll both fields into one product, which we do with the new recursor:

$$\texttt{iter}\{\texttt{z} \hookrightarrow e_0 \mid \texttt{s}(x) \hookrightarrow e_1\}(e)$$

We do not lose any expressive power with this construction, as we now only need to accumulate a pair whose first element is the predecessor (a number), and whose second element is the accumulated computation. In fact, we can build the old recursor directly:

$$\texttt{rec}\{\texttt{z} \hookrightarrow e_0 \mid \texttt{s}(x)\ \texttt{with}\ y \hookrightarrow e_1\}(e) \triangleq \texttt{iter}\{\texttt{z} \hookrightarrow (\texttt{z}, e_0) \mid \texttt{s}(x_1) \hookrightarrow (\texttt{s}(x_1 \cdot \texttt{l}), [x_1 \cdot \texttt{l}, x_1 \cdot \texttt{r}/x, y]e_1)\}(e) \cdot \texttt{r}$$

You should check that this construction is correct, which would mean that we can compute all the things we could with standard System **T**.

4 Type Isomorphisms

The fact that we have sum and product types brings up the question: just how similar are these types to addition and multiplication in arithmetic? One similarity that we can easily see is that simple equations in arithmetic also hold for types. For example, in arithmetic, we can say that $1 + 1 = 2$. For types, we have an equivalent notion: $1 + 1 \cong 2$. That is, $1 + 1$ is **isomorphic** to 2.

Two types are isomorphic precisely when they contain exactly the same information. For example, any value with type $1 \times \tau$ has no more information than a value of type τ. More formally, we say that $\tau_1 \cong \tau_2$ iff there are two functions $f : \tau_1 \to \tau_2$ and $g : \tau_2 \to \tau_1$ that are inverses of each other (such that $f(g(x)) = x$ and $g(f(x)) = x$).

We would like some ordinary properties of arithmetic to hold for sums and products. For example, intuitively, the following should hold:

$$\tau \times 0 \cong 0$$
$$\tau \times 1 \cong \tau$$
$$\tau_1 + \tau_2 \cong \tau_2 + \tau_1$$

We can show that these properties hold by writing some functions:

$\tau \times 0 \cong 0$:

$$\mathtt{f} \triangleq \lambda\,(x : \tau \times 0)\, x \cdot \mathtt{r}$$
$$\mathtt{g} \triangleq \lambda\,(x : 0)\, \mathtt{case}\ x\ \{\}$$

$\tau \times 1 \cong \tau$:

$$\mathtt{f} \triangleq \lambda\,(x : \tau \times 1)\, x \cdot \mathtt{l}$$
$$\mathtt{g} \triangleq \lambda\,(x : \tau)\, \langle x, () \rangle$$

$\tau_1 + \tau_2 \cong \tau_2 + \tau_1$:

$$\mathtt{f} \triangleq \lambda\,(x : \tau_1 + \tau_2)\, \mathtt{case}\ x\ \{\mathtt{l} \cdot x \hookrightarrow \mathtt{r} \cdot x \mid \mathtt{r} \cdot x \hookrightarrow \mathtt{l} \cdot x\}$$
$$\mathtt{g} \triangleq \lambda\,(x : \tau_2 + \tau_1)\, \mathtt{case}\ x\ \{\mathtt{l} \cdot x \hookrightarrow \mathtt{r} \cdot x \mid \mathtt{r} \cdot x \hookrightarrow \mathtt{l} \cdot x\}$$

As an exercise, show that each `f` and `g` are inverses. The first one is a bit tricky: the key insight is to realize that there are no values of types $\tau + 0$ or 0.

5 Generic Programming

A **type operator** is a type abstracted with a variable whose occurrences mark the spots in the type where a transformation will be applied. It is an abstractor $t.\tau$ such that t $\mathsf{type} \vdash \tau$ type. We will be looking at *polynomial type operators* in particular. These are simply type operators that are constructed from `unit`, `void`, and sums and products.

For example,

$$t.\mathtt{unit} + (\mathtt{bool} \times t)$$

is a polynomial type operator.

Type operators give us the ability to take a term and transform every subterm of a particular type τ into a new subterm of some type τ'. The operation that does this is called `map`. For example, say we have the following expression

$$\langle \mathtt{l} \cdot (), \mathtt{r} \cdot () \rangle : \mathtt{bool} \times \mathtt{bool}$$

and we want to flip all the booleans in it: turn `false` ($\mathtt{r} \cdot ()$) to `true` ($\mathtt{l} \cdot ()$) and vice versa. To do this, we first create a type operator that marks all the spots in the type where there are booleans:

$$t.t \times t$$

and then apply `map` with an transformation that flips booleans. This will apply that transformation at every t in the type operator, correctly flipping all the booleans:

$$\mathtt{map}\{t.t \times t\}(x.\mathtt{case}\ x\ \{\mathtt{l} \cdot \_ \hookrightarrow \mathtt{r} \cdot () \mid \mathtt{r} \cdot \_ \hookrightarrow \mathtt{l} \cdot ()\})(e) \longmapsto \langle \mathtt{r} \cdot (), \mathtt{l} \cdot () \rangle$$

More formally, the dynamics and statics of `map` are given below:

5.1 Statics

$$\frac{t.\tau\ \mathsf{poly} \quad \Gamma, x : \rho \vdash e' : \rho' \quad \Gamma \vdash e : [\rho/t]\tau}{\Gamma \vdash \mathtt{map}\{t.\tau\}(x.e')(e) : [\rho'/t]\tau}$$

5.2 Dynamics

$$\frac{}{\mathtt{map}\{t.t\}(x.e')(e) \longmapsto [e/x]e'}$$

$$\frac{}{\mathtt{map}\{t.\mathtt{unit}\}(x.e')(e) \longmapsto e}$$

$$\frac{}{\mathtt{map}\{t.\tau_1 \times \tau_2\}(x.e')(e) \longmapsto \langle \mathtt{map}\{t.\tau_1\}(x.e')(e \cdot \mathtt{l}), \mathtt{map}\{t.\tau_2\}(x.e')(e \cdot \mathtt{r}) \rangle}$$

$$\frac{}{\mathtt{map}\{t.\mathtt{void}\}(x.e')(e) \longmapsto \mathtt{case}\ e\ \{\}}$$

$$\frac{}{\mathtt{map}\{t.\tau_1 + \tau_2\}(x.e')(e) \longmapsto \mathtt{case}\ e\ \{\mathtt{l} \cdot x \hookrightarrow \mathtt{l} \cdot \mathtt{map}\{t.\tau_1\}(x.e')(x) \mid \mathtt{r} \cdot y \hookrightarrow \mathtt{r} \cdot \mathtt{map}\{t.\tau_2\}(x.e')(y)\}}$$

The dynamics for `map` specify based on the structure of the type operator which operations should be done on the input term e, with the result being that certain components of e get replaced according to e'. In the first rule, τ is simply t, meaning that the entirety of e gets replaced according to e', and we just substitute it in. In the second rule, the type operator has no instances of t, meaning that we must just return e untransformed. The third and fifth rules split e into components based on the product and sum rules, while the fourth rule expresses a paradoxical case with `void`. To understand the `void` rule, think about what it would mean for e to satisfy the type operator $t.\texttt{void}$, and what we could do to it at evaluation.

At this point you may be wondering: what is the purpose of generic programming? If we want to apply a transformation to an expression, why don't we just write code that does that? Why use `map`? We'll find out in lecture tomorrow that generic programming allows us to create powerful new types called inductive types. The essence is that given a specification of some type τ, we can automatically transform the type, so that our language's dynamics do not need to change to reflect new data structures.