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 $\tau_1$ and $\tau_2$, $\tau_1 \times \tau_2$, 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.

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 \texttt{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 \texttt{unit} conveys no real data, it has tremendous utility in programming languages.

Mainstream programming languages often conflate \texttt{unit} with \texttt{void}, a different concept altogether, often speaking of functions of “\texttt{void} return type.” In programming language theory we use the proper terminology, \texttt{unit}!

\textbf{Examples} of products:

\[
\begin{align*}
() & : \texttt{unit} \\
(z,z) & : \texttt{nat} \times \texttt{nat} \\
(z,(s(z),s(z))) & : \texttt{nat} \times (\texttt{nat} \times \texttt{nat}) \\
(\lambda (x: \texttt{nat}) x, \lambda (x: \texttt{nat} \rightarrow \texttt{nat}) x) & : (\texttt{nat} \rightarrow \texttt{nat}) \times ((\texttt{nat} \rightarrow \texttt{nat}) \rightarrow \texttt{nat} \rightarrow \texttt{nat})
\end{align*}
\]

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

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

\[
(\tau_1, \tau_2, \tau_3)
\]

and various labeled forms

\[
\langle \text{left} \leftrightarrow \tau_1, \text{right} \leftrightarrow \tau_2 \rangle
\]

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 \textbf{projections} retrieve the left and right branches of a tuple in the natural way.

### 1.1 Statics

\[
\Gamma \vdash () : \texttt{unit} \quad 
\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad 
\Gamma \vdash e : \tau_1 \times \tau_2 \quad 
\Gamma \vdash e \cdot l : \tau_1 \quad 
\Gamma \vdash e \cdot r : \tau_2
\]

### 1.2 Dynamics

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

\[
\begin{align*}
() & \Rightarrow e_1 \Rightarrow e'_1 \\
(e_1, e_2) & \Rightarrow (e'_1, e_2) \\
\end{align*}
\]

\[
\begin{align*}
e_1 \Rightarrow e'_1 \\
(e_1, e_2) & \Rightarrow (e'_2, e'_2) \\
\end{align*}
\]

\[
\begin{align*}
(e_1, e_2) \cdot l & \Rightarrow e_1 \\
(e_1, e_2) \cdot r & \Rightarrow e_2
\end{align*}
\]
2 Sums

Sum types, denoted $\tau_1 + \tau_2$, are a tagged disjoint union of the types $\tau_1$ and $\tau_2$. That is, a value of type $\tau_1 + \tau_2$ contains either a value of type $\tau_1$ or a value of type $\tau_2$, 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:

```
int + string + 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 $\mathbf{T}$ with sums:

```
Typ $\tau$ ::= ...  
  void      void  
  $\tau_1 + \tau_2$ sum

Exp $e$ ::= ...  
  $l\{\tau_1;\tau_2\} \cdot e$ left injection  
  $r\{\tau_1;\tau_2\} \cdot e$ right injection  
  case $e$ \{ $l \cdot x_1 \leftrightarrow e_1$ | $r \cdot x_2 \leftrightarrow e_2$ \} 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:

```
$1 \cdot e$  $r \cdot e$
```
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:

\[
\begin{align*}
&1 \cdot z : \text{nat} + \tau \\
&r \cdot s(z) : \tau + \text{nat} \\
&1 \cdot r \cdot \lambda(x : \text{nat}) x : (\tau_1 + (\text{nat} \to \text{nat})) + \tau_2
\end{align*}
\]

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 $\tau$, 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:

\[
\text{case } l\{\text{nat} \to \text{nat}; \text{nat}\} \cdot \lambda(x : \text{nat}) x \{1 \cdot x_1 \leftarrow x_1(z) \mid r \cdot x_2 \leftarrow x_2\}
\]

This case expression examines its operand, $l\{\text{nat} \to \text{nat}; \text{nat}\} \cdot \lambda(x : \text{nat}) x$, and in the left case binds its wrapped value to $x_1$ in $x_1(z)$. 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:

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

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:

\[
\lambda(x : \text{void}) \text{ case } x \{\}
\]

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, 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:

\[
\begin{align*}
\text{bool} &\triangleq \text{unit} + \text{unit} \\
\text{true} &\triangleq 1 \cdot () \\
\text{false} &\triangleq r \cdot ()
\end{align*}
\]

### 2.1 Statics

\[
\frac{
\Gamma \vdash e : \text{void}
}{
\Gamma \vdash \text{case } e \{\} : \tau}
\]

\[
\frac{
\Gamma \vdash e : \tau_1 \\
\Gamma \vdash l\{\tau_1; \tau_2\} \cdot e : \tau_1 + \tau_2
}{
\Gamma \vdash \text{case } e \{1 \cdot x_1 \leftarrow e_1 \mid r \cdot x_2 \leftarrow e_2\} : \tau}
\]

\[
\frac{
\Gamma \vdash e : \tau_1 + \tau_2 \\
\Gamma, x_1 : \tau_1 \vdash e_1 : \tau \\
\Gamma, x_2 : \tau_2 \vdash e_2 : \tau
}{
\Gamma \vdash \text{case } e \{1 \cdot x_1 \leftarrow e_1 \mid r \cdot x_2 \leftarrow e_2\} : \tau}
\]
2.2 Dynamics

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

\[
\frac{e \mapsto e'}{\text{case } e \{ \} \mapsto \text{case } e' \{ \}}
\]

\[
\frac{e \text{ val}}{\text{val} \cdot e \text{ val}} \quad \frac{e \text{ val}}{\text{val} \cdot e \text{ val}}
\]

\[
\frac{e \mapsto e'}{\text{1} \cdot e \mapsto \text{1} \cdot e'} \quad \frac{e \mapsto e'}{\text{r} \cdot e \mapsto \text{r} \cdot e'}
\]

\[
\frac{e \mapsto e'}{\text{case } e \{ \text{1} \cdot x_1 \mapsto e_1 | \text{r} \cdot x_2 \mapsto e_2 \} \mapsto \text{case } e' \{ \text{1} \cdot x_1 \mapsto e_1 | \text{r} \cdot x_2 \mapsto e_2 \}}
\]

\[
\frac{e \text{ val}}{\text{case } \text{1} \cdot e \{ \text{1} \cdot x_1 \mapsto e_1 | \text{r} \cdot x_2 \mapsto e_2 \} \mapsto [e/x_1]e_1}
\]

\[
\frac{e \text{ val}}{\text{case } \text{r} \cdot e \{ \text{1} \cdot x_1 \mapsto e_1 | \text{r} \cdot x_2 \mapsto e_2 \} \mapsto [e/x_2]e_2}
\]

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:

\[
\text{iter} \{ z \mapsto e_0 | s(x) \mapsto e_1 \}(e)
\]

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:

\[
\text{rec} \{ z \mapsto e_0 | s(x) \text{ with } y \mapsto e_1 \}(e) \triangleq \text{iter} \{ z \mapsto (z,e_0) | s(x_1) \mapsto (s(x_1 \cdot \text{1}), [x_1 \cdot \text{1}, x_1 \cdot \text{r}/x, y] e_1) \}(e) \cdot \text{r}
\]

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 \equiv 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 \(\tau\). More
formally, we say that \(\tau_1 \equiv \tau_2\) iff there are two functions \(f: \tau_1 \rightarrow \tau_2\) and \(g: \tau_2 \rightarrow \tau_1\) 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:
\[
\begin{align*}
\tau \times 0 & \cong 0 \\
\tau \times 1 & \cong \tau \\
\tau_1 + \tau_2 & \cong \tau_2 + \tau_1
\end{align*}
\]

We can show that these properties hold by writing some functions:
\[
\begin{align*}
\tau_1 \times 0 & \cong 0: \\
\tau_1 \times 1 & \cong \tau:
\end{align*}
\]

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 \(\tau \times 0\) or 0.