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: \textbf{product} and \textbf{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 $\mathbf{T}$, augmented with product types. Note the slightly different recursor, which now only binds one predecessor. We’ll get back to that in a moment.

\[
\begin{align*}
\textbf{Typ} \quad \tau & := \text{nat} \quad \text{number} \\
& \quad \tau_1 \to \tau_2 \quad \text{function} \\
& \quad \text{unit} \quad \text{unit} \\
& \quad \tau_1 \times \tau_2 \quad \text{product} \\
\textbf{Exp} \quad e & := x \quad \text{variable} \\
& \quad z \quad \text{zero} \\
& \quad \texttt{s}(e) \quad \text{successor} \\
& \quad \texttt{iter}\{z \leftrightarrow e_0 \mid \texttt{s}(x) \leftrightarrow e_1\}(e) \quad \text{recursion} \\
& \quad \lambda(x : \tau)e \quad \text{abstraction} \\
& \quad (e_1) \quad \text{application} \\
& \quad \langle \rangle \quad \text{empty pair} \\
& \quad \langle e_1, e_2 \rangle \quad \text{pair} \\
& \quad e \cdot l \quad \text{left projection} \\
& \quad e \cdot r \quad \text{right projection}
\end{align*}
\]

This is a language with binary products. The values of product type are created by the \textbf{pair} and \textbf{empty pair} constructors, their \textit{introduction forms}, and they are converted back into their constituent types by the \textbf{left} and \textbf{right projections}, their \textit{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}!

**Examples** of products:

\[
\langle \rangle : \texttt{unit} \\
\langle z, z \rangle : \texttt{nat} \times \texttt{nat} \\
\langle z, (s(z), s(z)) \rangle : \texttt{nat} \times (\texttt{nat} \times \texttt{nat}) \\
\langle \lambda (x : \texttt{nat}) x, \lambda (x : \texttt{nat} \rightarrow \texttt{nat}) x \rangle : (\texttt{nat} \rightarrow \texttt{nat}) \times ((\texttt{nat} \rightarrow \texttt{nat}) \rightarrow \texttt{nat} \rightarrow \texttt{nat})
\]

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

\[
\langle \tau_1, \tau_2, \tau_3 \rangle
\]

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

### 1.1 Statics

\[
\Gamma \vdash \langle \rangle : \texttt{unit} \\
\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2 \\
\Gamma \vdash e : \tau_1 \rightarrow \tau_1 \\
\Gamma \vdash e \cdot \text{\texttt{l}} : \tau_2
\]

### 1.2 Dynamics

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

\[
\langle \rangle \text{ val} \quad \langle e_1, e_2 \rangle \text{ val} \\
\text{ e_1 val } e_1' \quad e_2 \text{ val } e_2' \\
\text{ (e_1, e_2) val } (e_1', e_2') \\
\text{ e_1 val } e_2 \text{ val } (e_1, e_2) \text{ l } e_1 \\
\text{ e_1 val } e_2 \text{ val } (e_1, e_2) \text{ r } e_2
\]


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:

\[
\text{datatype token} = \text{Number of int} \mid \text{Identifier of string} \mid \text{Semicolon}
\]

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:

\[ \text{int + string + unit} \]

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 T with sums:

\[
\begin{align*}
\text{Typ} & \quad \tau \ ::= \ldots \\
& \quad \text{void} \quad \text{void} \\
& \quad \tau_1 + \tau_2 \quad \text{sum} \\
\text{Exp} & \quad e ::= \ldots \\
& \quad 1\{\tau_1;\tau_2\} \cdot e \quad \text{left injection} \\
& \quad r\{\tau_1;\tau_2\} \cdot e \quad \text{right injection} \\
& \quad \text{case } e \{1 \cdot x_1 \mapsto e_1 \mid r \cdot x_2 \mapsto e_2\} \quad \text{case}
\end{align*}
\]

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:

\[ 1 \cdot e \quad r \cdot e \]
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:

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

This `case` expression examines its operand, $1\{\text{nat} \to \text{nat}; \text{nat}\} \cdot \lambda(x : \text{nat})x$, 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:

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

### 2.1 Statics

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

\[
\begin{align*}
\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash 1\{\tau_1; \tau_2\} \cdot e : \tau_1 + \tau_2}{\Gamma \vdash \text{case } e \{\} : \tau} \\
\frac{\Gamma \vdash e : \tau_2 \quad \Gamma \vdash r\{\tau_1; \tau_2\} \cdot e : \tau_1 + \tau_2}{\Gamma \vdash \text{case } e \{\} : \tau}
\end{align*}
\]

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

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

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

\[
\frac{\text{e val}}{\begin{array}{cc}
& e \rightarrow e' \\
1 \cdot e & \rightarrow 1 \cdot e' \\
r \cdot e & \rightarrow r \cdot e'
\end{array}}
\]

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

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

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

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:

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

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:

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

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 Isomorphism

In lectures, we saw how a lot of interesting types we see everyday can be implemented using sums and products. For example, we define `bool` as `unit + unit`, and define `τ opt` to be `unit + τ`. Notice that it would not make any difference if we instead defined `τ opt` to be `τ + unit`. Obviously it’s equivalent to `unit + τ` in the sense that if we are given a value with type `unit + τ`, we can immediately generate a value with type `τ + unit`.

In our terminology, we say that these two types are isomorphic to each other, and use the following notation: `unit + τ ∼= τ + unit`. Formally, two types `τ₁, τ₂` in a language are isomorphic if there are two mutually inverse functions `e : τ₁ → τ₂` and `e : τ₂ → τ₁`. Informally, that means we can convert values of isomorphic types back and forth. Here are some examples of isomorphic types:

\[
\begin{align*}
τ & \cong τ \\
τ₁ + τ₂ & \cong τ₂ + τ₁ \\
τ₁ × τ₂ & \cong τ₂ × τ₁ \\
τ + void & \cong τ \\
τ × unit & \cong τ \\
τ × 2 & \cong τ + τ \\
τ × void & \cong void
\end{align*}
\]

These examples explain why we denote `unit` as 1 and `void` as 0: because they are the identities for products and sums, respectively, with respect to type isomorphism. Intuitively, two types are isomorphic if their values carry the same kind and amount of information. For example, if someone gives us a value of type `τ × unit`, then by definition we automatically know that the right projection is `{}`. So the only non-trivial information is the `τ`. Similarly, if someone gives us a `τ + void`, we automatically know that it must be a left injection, because there is no value of type `void`. So the "+void" part gives no extra information.

To formally prove, for example, `τ + void ∼= τ`, we explicitly write out a function of type `τ + void → τ`:

\[
λ(x : τ + void) \ case \ x \{ 1 · x₁ \mapsto x₁ | r · x₂ \mapsto \ case \ x₂ \{ \} \}
\]

The other direction is similar and left as an exercise. Notice that to show two types are isomorphic, your two functions have to be inverse of each other. In other words, if you evaluate `e₂(e₁(x))`, you should get back `x`. 