Overview

Modern programming languages include more interesting, and more complex, data types than just and.*

To analyze such features, we'll start with these types:

- **Product, or tuple, types.**

- **Sum, or disjoint union, types.**

Product Types

Product, or tuple, types give you structured data.

- Nullary products: unit. Sole value is {}.

- Binary products: $\tau_1 \times \tau_2$. Values are ordered pairs.

- $n$-ary products: $\Pi_{i \in I} \tau_i$. Values are ordered $n$-tuples.

- Labelled products, or records: `{name: string, salary: float}`. Elements are labelled tuples. Records are a basis for objects.

Product Types: Abstract and Concrete Syntax

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>$\tau$</td>
<td>$\text{unit}$</td>
<td>$\text{unit}$</td>
</tr>
<tr>
<td></td>
<td>$\tau_1 \times \tau_2$</td>
<td>$\tau_1 \times \tau_2$</td>
<td>$\text{triv}$</td>
</tr>
<tr>
<td>Expr</td>
<td>$e$</td>
<td>$\text{pair}(\tau_1; \tau_2)$</td>
<td>$\text{pair}(\tau_1; \tau_2)$</td>
</tr>
<tr>
<td></td>
<td>$e \vdash \tau_1$</td>
<td>$\text{proj}<a href="e">l</a>$</td>
<td>$\text{proj}<a href="e">l</a>$</td>
</tr>
<tr>
<td></td>
<td>$e \vdash \tau_2$</td>
<td>$\text{proj}<a href="e">r</a>$</td>
<td>$\text{proj}<a href="e">r</a>$</td>
</tr>
</tbody>
</table>

Binary (and nullary) product types.

Introductory form is pairing (**unit element** or null **tuple**).

Eliminatory form is projection (none for nullary).

Product Types: Static Semantics

$$
\Gamma \vdash \text{triv} : \text{unit}
$$

$$
\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2
\Rightarrow
\Gamma \vdash \text{pair}(e_1; e_2) : \tau_1 \times \tau_2
$$

$$
\Gamma \vdash e : \tau_1 \times \tau_2
\Rightarrow
\Gamma \vdash \text{proj}[l](e) : \tau_1
$$

$$
\Gamma \vdash e : \tau_1 \times \tau_2
\Rightarrow
\Gamma \vdash \text{proj}[r](e) : \tau_2
$$

Product Types: Dynamic Semantics

$$
\{e_1 \text{ val}, e_2 \text{ val}\}
\text{pair}(e_1; e_2) \text{ val}
$$

$$
\{e_1 \rightsquigarrow e'_1
\text{pair}(e_1; e_2) \rightsquigarrow \text{pair}(e'_1; e_2)\}
$$

$$
\{e_2 \rightsquigarrow e'_2
\text{pair}(e_1; e_2) \rightsquigarrow \text{pair}(e_1; e'_2)\}
$$

Bracketed premises and rules are omitted for lazy semantics and included for eager semantics of pairing.
Bracketed premises are omitted for lazy semantics and included for eager semantics of pairing.

Finite Product Types: Abstract and Concrete Syntax

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>τ</td>
<td>::= prod(f)(i =&gt; τ_i) (\cap_{i \in I} \tau_i)</td>
<td>(\cap_{i \in I} \tau_i)</td>
</tr>
<tr>
<td>Expr</td>
<td>e</td>
<td>::= tuple(f)(i =&gt; e_i) (\in\prod_{i \in I} \tau_i)</td>
<td>(\in\prod_{i \in I} \tau_i)</td>
</tr>
</tbody>
</table>

Finite Products: Static Semantics

\[(\forall i \in I) \Gamma \vdash e_i : \tau_i\]
\[\Gamma \vdash \text{tuple}(f)(i => e_i) : \text{prod}(f)(i => \tau_i)\]
\[\Gamma \vdash e : \text{prod}(f)(i => \tau_i) \quad j \in I\]
\[\Gamma \vdash \text{proj}(f)(i)(e) : \tau_i\]

Finite Products: Dynamic Semantics

\[\{\forall i \in I\} \{e_i : \tau_i\} \quad \text{tuple}(f)(i => e_i) \quad \text{val}\]
\[\text{proj}(f)(i)(e) \quad \text{val}\]

Finite Product Types: Safety

**Theorem 1**
1. If \(e : \tau\) and \(e \Rightarrow e'\), then \(e' : \tau\).
2. If \(e : \tau\), then either \(e\) val, or there exists \(e'\) such that \(e \Rightarrow e'\).

**Preservation**: By induction on evaluation.

**Progress**: By induction on typing. Canonical forms of product type are pairs. Can always project from a pair of the right type.

Grammar is indexed by a finite index \(I\) of size \(n\), such that \(\text{prod}(f)(i => \tau_i)\) is an \(n\)-argument operator of arity \((0, \ldots, 0)\) whose \(i\)th argument is type \(\tau_i\) \(\cap_{i \in I} \tau_i\).

Similarly, \(\text{tuple}(f)(i => e_i)\) is an \(n\)-argument abt operator of arity \((0, \ldots, 0)\) whose \(i\)th operand is \(e_i\) \(\in\prod_{i \in I} \tau_i\).

Projections are indexed by a constant \(0 \leq i < n\) indicating position to select from \(n\)-tuple.

Introductory form is **tupling**.

Eliminatory form is **(indexed) projection**.

Bracketed rule omitted for lazy semantics and included for eager semantics.
Safety for Finite Products

Theorem 2

1. If \( e : \tau \) and \( e \mapsto e' \), then \( e' : \tau \).

2. If \( e : \tau \), then either \( e \) is a value, or there exists \( e' \) such that \( e \mapsto e' \).

or equivalently:

If \( e : \tau \), then either \( e \) is a value, or there exists \( e' \) such that \( e' : \tau \) and \( e \mapsto e' \).

Special Cases of Finite Products

- Nullary products: \( \text{unit} = \{ \} \equiv \{ \} \)

- Binary products: \( \tau_1 \times \tau_2 = \{ (e_1, e_2) | \langle e_1, e_2 \rangle = \langle \langle 1 \rangle, \langle 2 \rangle \rangle : \tau_1 \times \tau_2 \} \)

pr\(_1\)(e) = \( e \) : \( 1 \) ; pr\(_2\)(e) = \( e \) : \( 2 \)

- Labelled products (records): Given a set \( L = \{ l_0, \ldots, l_{n-1} \} \) of

field names or field labels, product type \( \Pi_{l_0 : \omega_0, \ldots, l_{n-1} : \omega_{n-1}} \)

has values \( \langle d_0, \ldots, d_{n-1} : \omega_{n-1} \rangle \) with \( d_i : \omega_i \) for \( 0 \leq i < n \) and

and the projection \( e : l \) returns the component of \( e \) labelled by

\( l \in L \).

Sum Types

Sum, or disjoint union, types give you choices.

- Nullary: \( \text{void} \), with no elements.

- Binary: \( \tau_1 + \tau_2 \). Values are either a value of type \( \tau_1 \) tagged

\( \text{ia}[1] \), or a value of type \( \tau_2 \) tagged \( \text{ia}[r] \).

- N-ary: \( \tau_1 + \cdots + \tau_n \).

- Labelled: \([\text{present:string}, \text{absent:unit}]\).

Sums: Informal Description

The type \( \tau_1 + \tau_2 \) is the disjoint union of \( \tau_1 \) and \( \tau_2 \).

- Values of each type \( \tau_1 \) and \( \tau_2 \) are included within it.

- Elements are tagged with \( \text{ia}[1] \) or \( \text{ia}[r] \) to indicate where

they came from.

Thus \( \text{ia} + \text{ia} \) is quite different from \( \text{ia} \! \! \times \text{ia} \! \! \times \text{ia} \).

- Elements are \( \text{ia}[1](n) \) and \( \text{ia}[r](n) \).

- Disjoint union is different from ordinary set union!

SumTypes: Abstract and Concrete Syntax

<table>
<thead>
<tr>
<th>Cat</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>( \tau ) ::=</td>
<td>\text{void}</td>
<td>\text{void}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{sum}(\tau_1; \tau_2) )</td>
<td>( \tau_1 + \tau_2 )</td>
</tr>
<tr>
<td>Exp</td>
<td>( e ) ::=</td>
<td>\text{abort}(r)(e)</td>
<td>\text{abort}(r)(e)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{in}<a href="r">1</a>(e) )</td>
<td>( \text{in}<a href="r">1</a>(e) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{in}<a href="r">2</a>(e) )</td>
<td>( \text{in}<a href="r">2</a>(e) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{case}(e; x_1; e_1; x_2; e_2) )</td>
<td>( \text{case}(e; x_1; e_1; x_2; e_2) )</td>
</tr>
</tbody>
</table>

Binary (and nullary) sum types.

Introductory form is \textit{injection} (none for nullary).

Eliminatory form is \textit{case analysis} (\text{abort}(r)(e) for nullary).

Sums: Static Semantics

\[
\begin{align*}
\Gamma &\vdash e : \text{void} \\
\Gamma &\vdash e : \tau_1 \quad \tau = \text{sum}(\tau_1; \tau_2) \\
\Gamma &\vdash \text{in}[1](r)(e) : \tau \\
\Gamma &\vdash e : \tau_2 \quad \tau = \text{sum}(\tau_1; \tau_2) \\
\Gamma &\vdash \text{in}[2](r)(e) : \tau \\
\Gamma &\vdash e : \text{sum}(\tau_1; \tau_2) \quad \tau_1, \tau_2 : \tau \\
\Gamma &\vdash \text{case}(e; x_1; e_1; x_2; e_2) : \tau
\end{align*}
\]
Sums: Dynamic Semantics

\[
\begin{align*}
\Gamma, e & : \tau \\
\Gamma, e & : \text{val} \\
\Gamma, \text{in}[l] \text{(} e \text{)} & : \sigma \\
\Gamma, e & : \text{val} \\
\Gamma, \text{in}[r] \text{(} e \text{)} & : \sigma \\
\Gamma, e & : \text{val} \\
\Gamma, \text{case} \text{(} e \text{)} & : \sigma \\
\Gamma, e & : \text{val} \\
\Gamma, \text{case} \text{(} e \text{)} & : \sigma \\
\Gamma, \text{case} \text{(} e \text{)} & : \sigma \\
\end{align*}
\]

Bracketed premises and rules are omitted for lazy semantics and included for eager semantics.

Safety for Sums

Theorem 3

1. If \( e : \tau \) and \( e \Rightarrow e' \), then \( e' : \tau \).

2. If \( e : \tau \), then either \( e \text{ val} \), or there exists \( e' \) such that \( e \Rightarrow e' \).

- Canonical forms of type \( \text{sum}(\tau_1 \rightarrow \tau_2) \): \( \text{in}[1] \text{(} e \text{)} \) or \( \text{in}[2] \text{(} e \text{)} \).
- The exhaustiveness of \( \text{case} \) is crucial for progress!
Safety for Finite Sums

Theorem 4

1. If $e : r$ and $e \equiv e'$, then $e' : r$.
2. If $e : r$, then either $e$ is a value, or there exists $e'$ such that $e \equiv e'$.

Using Products and Sums: Unit and Void

The type $\text{unit}$ has one element, $\text{triv}$. The type $\text{void}$ has no elements! Consequently,

- If a function has type $\text{int} \rightarrow \text{void}$, it must not terminate for any argument.
- If a function has type $\text{int} \rightarrow \text{wait}$, it might return, but the result has to be $\text{triv}$.

(Some languages use $\text{void}$ when they mean $\text{wait}$ . . . )

Using Products and Sums: Option Types

Can also use sum types to define option types:

<table>
<thead>
<tr>
<th>Category Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type $\tau$</td>
<td>$::= \text{opt}(r)$</td>
<td>$\tau \text{opt}$</td>
</tr>
<tr>
<td>Expr $e$</td>
<td>$::= \text{null}$</td>
<td>$\text{null}$</td>
</tr>
<tr>
<td></td>
<td>$</td>
<td>\text{just}(e)$</td>
</tr>
<tr>
<td></td>
<td>$</td>
<td>\text{ifnull}(r)(e; e_1; e_2)$</td>
</tr>
</tbody>
</table>

Values of type $\text{opt}(r)$ represent "optional" values of type $r$. Introductory forms are $\text{null}$, meaning "no value" and $\text{just}(e)$, meaning a specified value of type $r$. Eliminatory form discriminates between the two possibilities.

Special Cases of Finite Sums

- Nullary sums: $\text{void} = \Sigma_{i \in \emptyset} ; \text{abort}_i(e) = \text{case}_i(0)$
- Binary sums: We take $I = \{l, r\}$. Then $\Sigma_{i \in I} : \text{in}[l](e) = \text{in}[l](e)$ and $\text{in}[r](e) = \text{in}[r](e)$; $\text{case}_i: \text{in}[l](e_1) \Rightarrow e_1 | \text{in}[r](e_2) \Rightarrow e_2 = \text{case}_i: \text{in}[l](e_1) \Rightarrow e_1 | e_i | e_I$
- $n$-ary sums: Index set $I = \{0, \ldots, n - 1\}$ for some $n > 0$.
- Labelled sums: Index set $I = \{0, \ldots, n - 1\}$ of labels that serve as symbolic indices for injections and symbolic names for cases.

Booleans: Abstract and Concrete Syntax

Simplest, most familiar example of a sum type is $\text{bool}$:

<table>
<thead>
<tr>
<th>Category Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type $\tau$</td>
<td>$::= \text{bool}$</td>
<td>$\text{bool}$</td>
</tr>
<tr>
<td>Expr $e$</td>
<td>$::= \text{tt}$</td>
<td>$\text{tt}$</td>
</tr>
<tr>
<td></td>
<td>$</td>
<td>\text{ff}$</td>
</tr>
<tr>
<td></td>
<td>$</td>
<td>\text{ifnull}(r)(e; e_1; e_2)$</td>
</tr>
</tbody>
</table>

Values of type $\text{bool}$ are $\text{tt}$ and $\text{ff}$.

Expression $\text{if}(e; e_1; e_2)$ branches on the value of $e : \text{bool}$.

Using Products and Sums

Type $\text{bool}$ is definable from binary sums and nullary products:

- $\text{bool} = \text{sum}($unit; unit$)$
- $\text{tt} = \text{in}[l]($bool$)(\text{triv})$
- $\text{ff} = \text{in}[r]($bool$)(\text{triv})$
- $\text{if}(e; e_1; e_2) = \text{case}(e; e_1; e_2)$, where $x_1 \# e_1$ and $x_2 \# e_2$ i.e., $\text{if}(e; e_1; e_2) = \text{case}(e; e_1; e_2)$
Using Products and Sums

The option type is **definable** from binary sums and nullary products!

- \( \text{opt}(r) = \text{sum}(\text{unit}; r) \)
- \( \text{null} = \text{inl}[\text{opt}(r)][\text{triv}] \)
- \( \text{just}(c) = \text{inl}[x][\text{opt}(r)](c) \)
- \( \text{ifnull}(r)(c; e_1; e_2; e_3) = \text{case}(c; e_1; e_2; e_3) \)

The Null Pointer

Many languages have a so-called **null pointer** or **null object**.

- The value **null** in Java.
- The cast (\( T \rightarrow 0 \)) in C.

The “null pointer” is used to model the **absence** of a value.

- Often as a default initial value for variables.
- As a “base case” for complex data structures.

The Null Pointer

The null pointer is a standard source of bugs.

- Null pointer exception in Java.
- Bus error in C.

Standard languages have no ability to track whether a pointer is null.

- Must check for null on each access.
- Explicit null checks do not change the type.

The Null Pointer

In ML there is a **type distinction** between

- A **genuine** value of type \( r \), and
- An **optional** value of type \( r \text{ option} \).

The key to this is the presence of **sum types**.

- Case analysis **changes the type** from \( r \text{ option} \) to \( r \).
- The type system tracks whether a value is present or not! There is no need for a **NONE** check!

Skeletal ML code for working with options:

```ml
fun dispatch (x : r option) =
  case x
  of NONE => e_0
  | SOME (x' : r) => e_1
```

Within \( e_1 \) the variable \( x' \) is **known** not to be “null”!

SML: datatype ‘a option = NONE | SOME of ‘a
The Null Pointer

Skeletal Java code for working with null pointers:

```java
if (x == null) {
  #1
} else {
  #2
}
```

Within #2 the type of `x` is still `Object` and might still (at some later point) be null!

A harder case:

```java
if (MyMethod(x)) {
  #3
} else {
  #4
}
```

The compiler cannot (in general) track that `MyMethod` returning `false` implies that `x` is non-null!

Summary

Products support structured data.

- Similar to `struct`'s in C, but with automatic allocation and no "pointers".

Sums support alternative data.

- Choice of two distinguishable alternatives.
- Case analysis propagates type change.