

Overview

Modern programming languages include more interesting, and more complex, data types than `num` and `str`.

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: $\prod_{i \in I} \tau_i$. 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

Category	Item	Abstract	Concrete
Type	τ	<code>::= unit</code>	<code>unit</code>
		<code>prod($\tau_1; \tau_2$)</code>	$\tau_1 \times \tau_2$
Expr	e	<code>::= triv</code>	<code>()</code>
		<code>pair($e_1; e_2$)</code>	<code><e_1, e_2></code>
		<code>proj[l](e)</code>	<code>pr_l(e)</code>
		<code>proj[r](e)</code>	<code>pr_r(e)</code>

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

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

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

$$\frac{\Gamma \vdash e : \text{prod}(\tau_1; \tau_2)}{\Gamma \vdash \text{proj}[l](e) : \tau_1}$$

$$\frac{\Gamma \vdash e : \text{prod}(\tau_1; \tau_2)}{\Gamma \vdash \text{proj}[r](e) : \tau_2}$$

Product Types: Dynamic Semantics

$$\frac{}{\text{triv val}}$$

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

$$\left\{ \frac{e_1 \mapsto e'_1}{\text{pair}(e_1; e_2) \mapsto \text{pair}(e'_1; e_2)} \right\}$$

$$\left\{ \frac{e_1 \text{ val} \quad e_2 \mapsto e'_2}{\text{pair}(e_1; e_2) \mapsto \text{pair}(e_1; e'_2)} \right\}$$

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

Product Types: Dynamic Semantics

$$\frac{e \mapsto e'}{\text{proj}[l](e) \mapsto \text{proj}[l](e')}$$

$$\frac{e \mapsto e'}{\text{proj}[r](e) \mapsto \text{proj}[r](e')}$$

$$\frac{\{e_1 \text{ val}\} \quad \{e_2 \text{ val}\}}{\text{proj}[l](\text{pair}(e_1; e_2)) \mapsto e_1}$$

$$\frac{\{e_1 \text{ val}\} \quad \{e_2 \text{ val}\}}{\text{proj}[r](\text{pair}(e_1; e_2)) \mapsto e_2}$$

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

6

Product Types: Safety

Theorem 1

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

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

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.

7

Finite Product Types: Abstract and Concrete Syntax

Category	Item	Abstract	Concrete
Type	τ	$::= \text{prod}[I](i \mapsto \tau_i)$	$\prod_{i \in I} \tau_i$
Expr	e	$::= \text{tuple}[I](i \mapsto e_i)$	$\langle e_i \rangle_{i \in I}$
		$ \text{proj}[I][i](e)$	$e \cdot i$

8

Finite Product Types: Abstract and Concrete Syntax

Grammar is indexed by a finite index I of size n , such that $\text{prod}[I](i \mapsto \tau_i)$ is an n -argument operator of arity $(0, \dots, 0)$ whose i th argument is type τ_i : $\prod \langle i_0 : \tau_0, \dots, i_{n-1} : \tau_{n-1} \rangle$.

Similarly, $\text{tuple}[I](i \mapsto e_i)$ is an n -argument abt operator of arity $(0, \dots, 0)$ whose i th operand is e_i : $\langle i_0 : e_0, \dots, i_{n-1} : e_{n-1} \rangle$.

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**.

9

Finite Products: Static Semantics

$$\frac{(\forall i \in I) \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \text{tuple}[I](i \mapsto e_i) : \text{prod}[I](i \mapsto \tau_i)}$$

$$\frac{\Gamma \vdash e : \text{prod}[I](i \mapsto \tau_i) \quad j \in I}{\Gamma \vdash \text{proj}[I][j](e) : \tau_j}$$

10

Finite Products: Dynamic Semantics

$$\frac{\{(\forall i \in I) e_i \text{ val}\}}{\text{tuple}[I](i \mapsto e_i) \text{ val}}$$

$$\left\{ \frac{e_j \mapsto e'_j \quad (\forall i \neq j) e'_i = e_i}{\text{tuple}[I](i \mapsto e_i) \mapsto \text{tuple}[I](i \mapsto e'_i)} \right\}$$

$$\frac{e \mapsto e'}{\text{proj}[I][j](e) \mapsto \text{proj}[I][j](e')}$$

$$\frac{\text{tuple}[I](i \mapsto e_i) \text{ val}}{\text{proj}[I][j](\text{tuple}[I](i \mapsto e_i)) \mapsto e_j}$$

Bracketed rule omitted for lazy semantics and included for eager semantics.

11

Safety for Finite Products

Theorem 2

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

or equivalently:

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

12

Special Cases of Finite Products

- Nullary products: $\text{unit} = \Pi_{e \in \emptyset} \emptyset$; $\langle \rangle = \langle \emptyset \rangle_{e \in \emptyset}$
- Binary products: $\tau_1 \times \tau_2 = \Pi_{i \in \{1,2\}} \tau_i$; $\langle e_1, e_2 \rangle = \langle e_i \rangle_{i \in \{1,2\}}$; $\text{pr}_1(e) = e \cdot 1$; $\text{pr}_2(e) = e \cdot 2$
- Labelled products (records): Given a set $L = \{l_0, \dots, l_{n-1}\}$ of **field names** or **field labels**, product type $\Pi \langle l_0 : \tau_0, \dots, l_{n-1} : \tau_{n-1} \rangle$ has values $\langle l_0 : e_0, \dots, l_{n-1} : e_{n-1} \rangle$ with $e_i : \tau_i$ for $0 \leq i < n$ and the projection $e \cdot l$ returns the component of e labelled by $l \in L$.

13

Sum Types

Sum, or **disjoint union**, types give you choices.

- Nullary: `void`, with **no** elements.
- Binary: $\tau_1 + \tau_2$. Values are **either** a value of type τ_1 tagged `in[l]`, **or** a value of type τ_2 tagged `in[r]`.
- n -ary: $\tau_1 + \dots + \tau_n$.
- Labelled: `[present:string, absent:unit]`.

14

SumTypes: Abstract and Concrete Syntax

Cat	Item	Abstract	Concrete
Type	τ	<code>void</code> <code>sum($\tau_1; \tau_2$)</code>	<code>void</code> $\tau_1 + \tau_2$
Expr	e	<code>abort[τ](e)</code> <code>in[l][τ](e)</code> <code>in[r][τ](e)</code> <code>case($e; x_1.e_1; x_2.e_2$)</code>	<code>abort$_{\tau}$(e)</code> <code>in[l](e)</code> <code>in[r](e)</code> <code>case $e\{in[l](x_1) \Rightarrow e_1 \mid in[r](x_2) \Rightarrow e_2\}$</code>

Binary (and nullary) sum types.

Introductory form is **injection** (none for nullary).

Eliminatory form is **case analysis** (`abort[τ](e)` for nullary).

15

Sums: Informal Description

The type $\tau_1 + \tau_2$ is the **disjoint union** of τ_1 and τ_2 .

- Values of each type τ_1 and τ_2 are included within it.
- Elements are **tagged** with `in[l]` or `in[r]` to indicate where they came from.

Thus `int+int` is quite different from `int`!

- Elements are `in[l](n)` and `in[r](n)`.
- Disjoint union is different from ordinary set union!

16

Sums: Static Semantics

$$\frac{}{\Gamma \vdash e : \text{void}}$$

$$\frac{}{\Gamma \vdash \text{abort}[\tau](e) : \tau}$$

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau = \text{sum}(\tau_1; \tau_2)}{\Gamma \vdash \text{in}[\text{l}][\tau](e) : \tau}$$

$$\frac{\Gamma \vdash e : \tau_2 \quad \tau = \text{sum}(\tau_1; \tau_2)}{\Gamma \vdash \text{in}[\text{r}][\tau](e) : \tau}$$

$$\frac{\Gamma \vdash e : \text{sum}(\tau_1; \tau_2) \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \text{case}(e; x_1.e_1; x_2.e_2) : \tau}$$

17

Sums: Dynamic Semantics

$$\frac{e \mapsto e'}{\text{abort}[\tau](e) \mapsto \text{abort}[\tau](e')}$$

$$\frac{\{e \text{ val}\}}{\text{in}[\mathbf{l}][\tau](e) \text{ val}}$$

$$\frac{\{e \text{ val}\}}{\text{in}[\mathbf{r}][\tau](e) \text{ val}}$$

$$\left\{ \frac{e \mapsto e'}{\text{in}[\mathbf{l}][\tau](e) \mapsto \text{in}[\mathbf{l}][\tau](e')} \right\}$$

$$\left\{ \frac{e \mapsto e'}{\text{in}[\mathbf{r}][\tau](e) \mapsto \text{in}[\mathbf{r}][\tau](e')} \right\}$$

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

18

Sums: Dynamic Semantics

$$\frac{e \mapsto e'}{\text{case}(e; x_1.e_1; x_2.e_2) \mapsto \text{case}(e'; x_1.e_1; x_2.e_2)}$$

$$\frac{\{e \text{ val}\}}{\text{case}(\text{in}[\mathbf{l}][\tau](e); x_1.e_1; x_2.e_2) \mapsto [e/x_1]e_1}$$

$$\frac{\{e \text{ val}\}}{\text{case}(\text{in}[\mathbf{r}][\tau](e); x_1.e_1; x_2.e_2) \mapsto [e/x_2]e_2}$$

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

19

Safety for Sums

Theorem 3

1. If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.
 2. If $e : \tau$, then either $e \text{ val}$, or there exists e' such that $e \mapsto e'$.
- Canonical forms of type $\text{sum}(\tau_1; \tau_2) : \text{in}[\mathbf{l}][\tau](e)$ or $\text{in}[\mathbf{r}][\tau](e)$.
 - The exhaustiveness of `case` is crucial for progress!

20

Finite Sum Types: Abstract and Concrete Syntax

Category	Item	Abstract	Concrete
Type	τ	$::= \text{sum}[I](i \mapsto \tau_i)$	$\sum_{i \in I} \tau_i$
Expr	e	$::= \text{in}[I][j](e)$ $\text{case}[I](e; i \mapsto x_i.e_i)$	$\text{in}[j](e)$ $\text{case } e \{ \text{in}[i](x_i) \Rightarrow e_i \}_{i \in I}$

Grammar is indexed by a finite index I of size n , such that $\text{sum}[I](i \mapsto \tau_i)$ is an n -argument operator of arity $(0, \dots, 0)$ whose i th argument is type τ_i : $\Sigma \langle i_0 : \tau_0, \dots, i_{n-1} : \tau_{n-1} \rangle$, where $I = \{i_0, \dots, i_{n-1}\}$.

Both abstractors $(x_i.e_i)$ and injection labels $(\text{in}[I][j])$ are also I -indexed.

21

Finite Sums: Static Semantics

$$\frac{\Gamma \vdash e : \tau_j \quad j \in I}{\Gamma \vdash \text{in}[I][j](e) : \text{sum}[I](i \mapsto \tau_i)}$$

$$\frac{\Gamma \vdash e : \text{sum}[I](i \mapsto \tau_i) \quad (\forall i \in I) \Gamma, x_i : \tau_i \vdash e_i : \tau}{\Gamma \vdash \text{case}[I](e; i \mapsto x_i.e_i) : \tau}$$

22

Finite Sums: Dynamic Semantics

$$\frac{\{e \text{ val}\}}{\text{in}[I][j](e) \text{ val}}$$

$$\left\{ \frac{e \mapsto e'}{\text{in}[I][j](e) \mapsto \text{in}[I][j](e')} \right\}$$

$$\frac{e \mapsto e'}{\text{case}[I](e; i \mapsto x_i.e_i) \mapsto \text{case}[I](e'; i \mapsto x_i.e_i)}$$

$$\frac{\text{in}[I][j](e) \text{ val}}{\text{case}[I](\text{in}[I][j](e); i \mapsto x_i.e_i) \mapsto [e/x_j]e_j}$$

Bracketed premise and rule omitted for lazy semantics and included for eager semantics.

23

Safety for Finite Sums

Theorem 4

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 $e \mapsto e'$.

24

Special Cases of Finite Sums

- Nullary sums: $\text{void} = \Sigma_{-e\emptyset}^0$; $\text{abort}_r(e) = \text{case } e\{\emptyset\}$
- Binary sums: We take $I = \{l, r\}$. Then $\tau_1 + \tau_2 = \Sigma_{i \in I} \tau_i$; the injections $\text{in}[l](e) = \text{in}[l](e)$ and $\text{in}[r](e) = \text{in}[r](e)$; $\text{case } e\{\text{in}[l](x_1) \Rightarrow e_1 \mid \text{in}[r](x_2) \Rightarrow e_2\} = \text{case } e\{\text{in}[i](x_i) \Rightarrow e_i\}_{i \in I}$
- n-ary sums: Index set $I = \{0, \dots, n-1\}$ for some $n > 0$.
- Labelled sums: Index set $I = \{l_0, \dots, l_{n-1}\}$ of **labels** that serve as symbolic indices for injections and symbolic names for cases.

25

Using Products and Sums: Unit and Void

The type `unit` has **one** element, `triv`. The type `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 $\text{int} \rightarrow \text{unit}$, it **might return**, but the result has to be `triv`.

(Some languages use `void` when they mean `unit` ...)

26

Booleans: Abstract and Concrete Syntax

Simplest, most familiar example of a sum type is Booleans:

Category	Item	Abstract	Concrete
Type	τ	::= <code>bool</code>	<code>bool</code>
Expr	e	::= <code>tt</code> <code>ff</code> <code>if(e; e₁; e₂)</code>	<code>tt</code> <code>ff</code> <code>if e then e₁ else e₂</code>

Values of type `bool` are `tt` and `ff`.

Expression `if(e; e1; e2)` branches on the value of $e : \text{bool}$.

27

Using Products and Sums

Type `bool` is **definable** from binary sums and nullary products!

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

28

Using Products and Sums: Option Types

Can also use sum types to define **option types**:

Category	Item	Abstract	Concrete
Type	τ	::= <code>opt(τ)</code>	τ <code>opt</code>
Expr	e	::= <code>null</code> <code>just(e)</code> <code>ifnull[τ](e; e₁; x.e₂)</code>	<code>null</code> <code>just(e)</code> <code>check e\{null \Rightarrow e₁ just(x) \Rightarrow e₂\}</code>

Values of type `opt(τ)` represent "optional" values of type τ .

Introductory forms are `null`, meaning "no value" and `just(e)`, meaning a specified value of type τ . Eliminator form discriminates between the two possibilities.

29

Using Products and Sums

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

- $\text{opt}(\tau) = \text{sum}(\text{unit}; \tau)$
- $\text{null} = \text{in}[1][\text{opt}(\tau)](\text{triv})$
- $\text{just}(e) = \text{in}[r][\text{opt}(\tau)](e)$
- $\text{ifnull}[\tau](e; e_1; x_2.e_2) = \text{case}(e; _..e_1; x_2.e_2)$

30

The Null Pointer

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

- The value `null` in Java.
- The cast $(T *)0$ 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.

31

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.

32

The Null Pointer

But these problems never arise in ML! Why?

- Absence of “pointer mentality” — **value-oriented programming**.
- Without pointers there are no null pointers!

Why are there no null pointers in ML?

- Sum types obviate the need for them!
- SML: `datatype 'a option = NONE | SOME of 'a`

33

The Null Pointer

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

- A **genuine** value of type τ , and
- An **optional** value of type $\tau \text{ option}$.

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

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

34

The Null Pointer

Skeletal ML code for working with options:

```
fun dispatch (x :  $\tau$  option) =  
  case x  
  of NONE =>  $e_0$   
   | SOME ( $\overline{x' : \tau}$ ) =>  $e_1$ 
```

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

35

The Null Pointer

Skeletal Java code for working with null pointers:

```
if (x == null)
  s1
else
  s2
```

Within s_2 the type of x is still `Object` and might still (at some later point) be `null`!

36

The Null Pointer

A harder case:

```
if (MyMethod(x))
  s1
else
  s2
```

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

37

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.

38