Review: every enum value contains one piece of data

We call this "sum type" = enum on steroids

sum values

'R 1

" ϵ "

sum type

$+\{\underbrace{\text{'R}}_{\text{label}} : \underbrace{\text{Nat}}_{\text{data}},\ \text{'G} : \ldots,\ \text{'B} : \ldots\}$

introduction form (how to make a sum value?)

elimination form (how to use a sum value?)

~~switch~~ → (match)

```
switch e {
  'R x : ...
  'G y : ...
  'B z : ...
```

Pattern matching:

Combining the elimination forms of products and sums.

```
let (x, y, ... ) = e1 in
e2
```

```
switch e0 {
  'l1 x: e1,
  'l2 x: e2,
    ...
}
```

A pattern match is a list of branches.

- Branch = pat : expr.
- Each pat describes the expected "shape" of the data
- If the actual data matches the expected "shape", the expr is executed
- Go through branches sequentially to find & execute the first match

```
match e0 {
  pat1: e1,
  pat2: e2,
  ...
}
```

Pattern matching: Formal definition

```
expr    ::=  ...
          | (e1, .., en)              tuple
          | 'label e                  injection
          | match e { (branch)* }     pattern matching
value   ::=  ...
          | (v1, .., vn)              tuple value
          | 'label v                  injection value

branch ::= pat: expr
pat     ::=
          | (pat1, pat2, ... )        tuple pattern
          | 'label pat                injection pattern
          | x                         pattern variable
          | _                         wildcard
```

Operational semantics

(1st attempt)

Example: boolean negation

```
Bool = +{'true: (), 'false: ()}
match b with {
  'true (): 'false (),
  'false (): 'true ()
}
```

'true()

```
expr   ::= ...
         | (e1, .., en)               tuple
         | 'label e                   injection
         | match e { (branch)* }      pattern matching
value  ::= ...
         | (v1, .., vn)               tuple value
         | 'label v                   injection value

branch ::= pat: expr
pat    ::=
         | (pat1, pat2, ...)          tuple pattern
         | 'label pat                 injection pattern
         | x                          pattern variable
         | _                          wildcard
```

match (1,2) {
(x, y, z) : x
}

'true() 'true()

'false() $\not\bowtie$ 'true()

pat ⋈ **value** means actual **value** matches expected **pat**

$$\frac{}{x \bowtie v} \qquad \frac{}{\_ \bowtie v} \qquad \frac{('l = 'j) \quad pat \bowtie v}{'l\ pat \bowtie 'j\ v} \qquad \frac{p_1 \bowtie v_1 \quad p_2 \bowtie v_2 \quad \cdots \quad p_n \bowtie v_n}{(p_1, \ldots, p_n) \bowtie (v_1, \ldots, v_n)}$$

Operational semantics
(1st attempt)

Example: Nat decrement

```
Nat = +{'zero: (), 'succ: Nat}
match n with {
  'zero (): 'zero ()
  'succ m: m
}
```

value

$\boxed{\text{`succ ('zero ())}}$

```
expr   ::=  ...
          | (e1, .., en)              tuple
          | 'label e                  injection
          | match e { (branch)* }     pattern matching
value  ::=  ...
          | (v1, .., vn)              tuple value
          | 'label v                  injection value

branch ::= pat: expr
pat    ::=
          | (pat1, pat2, ... )        tuple pattern
          | 'label pat                injection pattern
          | x                         pattern variable
          | _                         wildcard
```

$$[\text{'zero ()}/m]\,(m) = \text{'zero ()}$$

$$\Downarrow$$

$$\text{'zero ()}$$

$$\frac{}{m \bowtie \text{'zero ()}}\ \text{var}$$

$$\frac{m \bowtie \text{'zero ()}}{\underbrace{\text{'succ}}_{\text{pat}}\ m \bowtie \underbrace{\text{'succ}}_{\text{value}}\ (\text{'zero ()})}$$

Operational semantics (2nd attempt)

$$\sigma ::= (\text{empty}) \mid \sigma,\ v/x$$

$$pat \bowtie value \to \sigma$$

Means **value** matches **pat** by using dictionary **σ** to map pattern variables to values

$$\frac{}{\lambda x.\, e \Rightarrow \lambda x.\, e}$$

$$\frac{}{x \bowtie v \to v/x}$$

$$\frac{}{\_ \bowtie v \to \cdot}$$

$$\frac{('l = 'j) \quad pat \bowtie v \to \sigma}{'l\ pat \bowtie 'j\ v \to \sigma}$$

$$\frac{p_1 \bowtie v_1 \to \sigma_1 \quad p_2 \bowtie v_2 \to \sigma_2 \quad \cdots \quad p_n \bowtie v_n \to \sigma_n}{(p_1, \ldots, p_n) \bowtie (v_1, \ldots, v_n) \to (\sigma_1, \sigma_2, \ldots, \sigma_n)}$$

$$\left(\lambda x.\, \mathrm{match}\ x\ \{\ \}\right)\ \underline{v}$$

Void

$$+\{\} = \text{"}F\text{"} = +\{\}$$

$$\to \quad \text{"}=\text{"} \quad \Rrightarrow\text{"}$$

$$F \Rightarrow F$$

Typing sums

injection

$$\underbrace{\texttt{+\{'num: Nat, 'bool: Bool\}}}_{\text{sum type}}$$

```
✓ 'num 3 : +{'num: Nat, 'bool: Bool}

✗ 'num 3 : +{'num: Bool, 'bool: Bool}

✓ 'num 3 : +{'num: Nat}

✗ 'num 3 : +{'nummm: Nat}
```

L can be any set of labels as long as $j \in L$.

$$\frac{(j \in L) \qquad \Gamma \vdash e : t_j}{\Gamma \vdash \underline{j}\ e : +\{\underline{l} : t_l\}_{l \in L}} \ \textsc{T-Inj}$$

Typing pattern matching: intuition

Q: for each match, will it get stuck during runtime?

```
Nat = +{'zero: (), 'succ: Nat}
n : Nat
```

```
match n with {
  'jeero (): 'zero ()
  'succ n: n
}
```

```
match n with {
  'zero (): 'zero ()
  'succ n: n
}
```

X

```
match n with {
  'jeero (): 'zero ()
  'succ n: n
}
```

missing 'zero ()

```
match (1,2) with {
  (x, y, z): x
}
```

eval will not get stuck, but programmer wrote sth dumb.

```
match n with {
  'succ n   : n,
  _         : 'zero (),
  'zero ()  : 'zero ()
}
```

```
match () with
{
  (): 1
}
```

X

```
match n with {
  'succ n: n
}
```

missing zero

```
Void = +{}
\x.
match x with {
}
```

Void → Void

Typing pattern matching: high-level ideas

```
match n with {
  'jeero (): 'zero ()
  'succ n: n
}
```

1. Check that in every branch, the pattern can match the type

Recall the operational semantics for pattern matching:

$$\texttt{pat} \bowtie \texttt{value} \to \sigma$$

$Nat = \{0, 1, 2, 3, \ldots\}$

means the `value` matches the `pat` using the substitution σ

Types = abstract values!

$$\texttt{pat} \bowtie \texttt{type} \to \Gamma$$

means at least one value in `type` matches the `pat`, using the substitution Γ

Typing pattern matching: high-level ideas

2. Check that all patterns collectively exhaust the values in a type

HW4: you will create your own design to check this

pat_1 : --

pat_2 : --

|

:

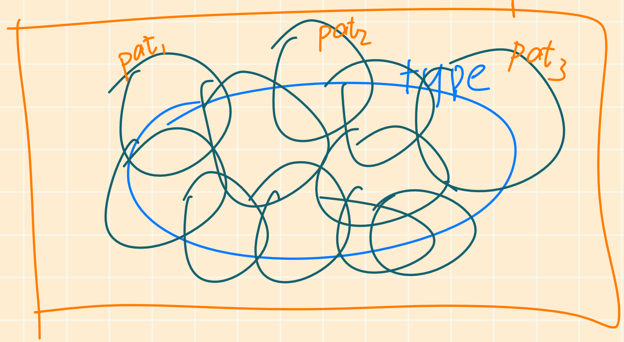
pat

```
match n with {
  'succ n: n
}
```

: 'zero ()

Think of type as a set

all possible values



Problem 1 (✍ written, 2 pts)

Finish the case of `e1 == e2` in the proof of the type abstraction lemma.

$\alpha(n) = Nat$

$\alpha(b) = Bool$

Lemma $\boxed{e : t \text{ implies } \boxed{e \Rightarrow v}}$ IH

Proof induct size(e)

$e_1 : Nat$

$e_1 \Rightarrow v_1 \wedge \alpha(v_1) = Nat$

n_1

· $e =$ "$e_1 == e_2$"

$e : t \Rightarrow$

$$\frac{\boxed{e_1 : Nat} \quad \boxed{e_2 : Nat}}{\boxed{e_1 == e_2} : Bool} \text{ T-Eq}$$

must hold

$e_2 : Nat$

$e_2 \Rightarrow v_2 \wedge \alpha(v_2) = Nat$

$\Rightarrow v_2 = n_2$

$$\frac{e_1 \Rightarrow n_1 \quad e_2 \Rightarrow n_2 \quad (n_1 == n_2)}{e_1 == e_2 \Rightarrow True} \text{ EqTrue}$$

$$\frac{}{\Rightarrow False} \text{ EqFalse.}$$

Problem 2 (✍ written, 2 pts)

Consider an alternative typing rule for `if` expressions:

```
Gamma |- e1 : Bool
Gamma |- e2 : t
---------------------------------- T-If*
Gamma |- if e1 then e2 else e3 : t
```

$\Gamma \vdash e_3 : t_3$

Redo the proof of the type abstraction lemma for `if` expressions using `T-If*` rule until the proof gets stuck. Then:

- Indicate the exact place where the proof fails
- Give some intuition why `T-If*` would not allow the proof to go through.
- Construct a counterexample that shows that a type system with this new rule is unsound.

counter-ex for type abstraction lemma

if False then 1 else True

if False then 1 else 1+True

conter-ex for type soundness theorem

$e1 : Bool$

$e_1 = True$

$e_1 = False$

$e_1 \Rightarrow False \quad \boxed{e_3 \Rightarrow v_3}$ This is not guaranteed.

if e_1 then e_2 else $e_3 \Rightarrow v_3$

Problem 3 (✍ written, 2 pts)

Pick one typing rule (other than the original `T-If` rule) for the variable-free subset of Lamp. Modify it to make the type system unsound. Then:

- Do the proof of the type abstraction lemma for this modified rule until the proof gets stuck.
- Indicate the exact place where the proof fails.
- Give some intuition why this modified typing rule would not allow the proof to go through.

$$\frac{e_1 : \mathrm{Nat}}{e_1 * e_2 : \mathrm{Nat}}$$

Problem 4 (✍️ written, 2 pts)

Construct a Lamp expression `e` such that

- `e` evaluates to a value
- `e` is ill-typed
- `e` does not use if-then-else. Essentially, this problem asks you to show that Lamp's type system is *incomplete* not only for if-then-else, but also for other constructs.

it's programmer's fault

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash (e : t) : t}$$

def X $\boxed{: \text{Nat}}$ = True

main : Bool = X

(1+2) : Nat

(1+2) : Bool

runs fine, but ill-typed.

Problem 5 (✍️ written, 2 pts)

Construct a Lamp expression `uncurry` that takes a function accepting "2 arguments", and returns an equivalent function that accepts "1 argument" that is a pair.

For example, consider the function `\x. \y. x + y`. Then `uncurry (\x. \y. x + y)` should return a function equivalent to the following:

```
\p. let (x, y) = p in x + y
```

uncurry: $\lambda f. \ \lambda p. \ \text{let } (x,y) = p \text{ in } f\ x\ y$

curry: $\lambda f. \ \lambda x. \lambda y. \ f(x,y)$

$$\dfrac{\dfrac{\dfrac{}{\cdots \vdash x : t_1}\ \text{Id}}{\cdots \vdash x : t_1}\ \text{Weaken} \qquad \dfrac{}{\cdots \vdash y : t_2}\ \text{Id}}{f : (t_1, t_2) \to t_3,\ x : t_1,\ y : t_2 \vdash (x, y) : (t_1, t_2)}\ \text{T-Pack}$$

$$\dfrac{\dfrac{\dfrac{f : (t_1, t_2) \to t_3 \qquad \vdash f : (t_1, t_2) \to t_3}{f : (t_1, t_2) \to t_3,\ x : t_1, \qquad \vdash f : (t_1, t_2) \to t_3}\ \text{Weaken}}{f : (t_1, t_2) \to t_3,\ x : t_1,\ y : t_2 \vdash f : (t_1, t_2) \to t_3}\ \text{Weaken}}{f : (t_1, t_2) \to t_3,\ x : t_1,\ y : t_2 \ \vdash \ f(x, y) : \ t_3}\ \text{T-App}$$

$$\dfrac{f : (t_1, t_2) \to t_3,\ x : t_1,\ y : t_2 \ \vdash \ f(x, y) : \ t_3}{f : (t_1, t_2) \to t_3,\ x : t_1 \ \vdash \ \lambda y.\ f(x, y) \ : \ t_2 \to t_3}\ \text{T-}\lambda$$

$$\dfrac{f : (t_1, t_2) \to t_3,\ x : t_1 \ \vdash \ \lambda y.\ f(x, y) \ : \ t_2 \to t_3}{f : (t_1, t_2) \to t_3 \vdash \lambda x.\ \lambda y.\ f(x, y) \ : \ t_1 \to (t_2 \to t_3)}\ \text{T-}\lambda$$

$$\dfrac{f : (t_1, t_2) \to t_3 \vdash \lambda x.\ \lambda y.\ f(x, y) \ : \ t_1 \to (t_2 \to t_3)}{\cdot \ \vdash \ \lambda f.\ \lambda x.\ \lambda y.\ f(x, y) \ : \ ((t_1, t_2) \to t_3) \to \ (t_1 \to (t_2 \to t_3))}\ \text{T-Lambda}$$

$$\text{size}(\text{Nat}) = \infty$$

$$\text{size}(\text{Bool}) = 2$$

$$\text{size}((t_1, t_2, \ldots, t_n)) = \begin{cases} \text{size}(t_1) \times \text{size}(t_2) \times \cdots \times \text{size}(t_n) & \text{if none of them is } \infty \\ \infty & \text{otherwise} \end{cases}$$

$$\text{size}(t_1 \to t_2) = \begin{array}{l} a \leftarrow \text{size}(t_1) \\ b \leftarrow \text{size}(t_2) \\ b^a \quad \text{if} \quad a \neq \infty ,\ b \neq \infty \\ \infty \quad \text{otherwise.} \end{array}$$