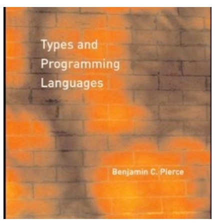
Simply-Typed λ-Calculus

Lecture 11
CS 565

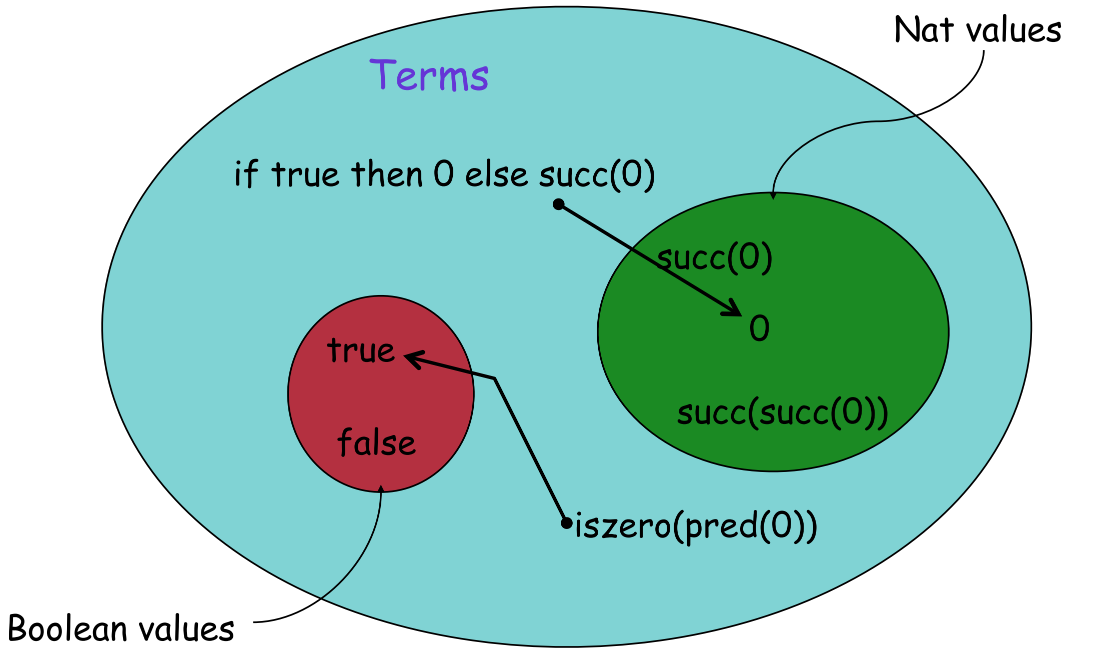


Boolean and Nat terms

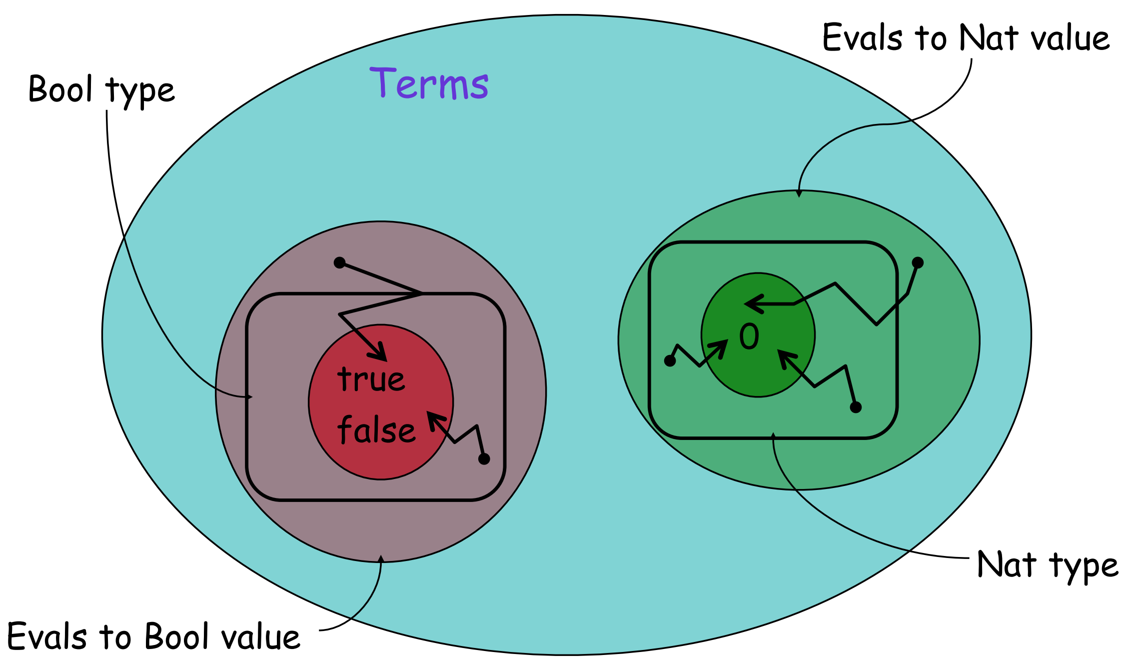
Some terms represent booleans, some represent natural numbers.

```
t ::= true
      false
      if t then t else t    ->  if t then t else t
                            ->  if t then t else t
      0
      succ t
      pred t
      iszero t
```

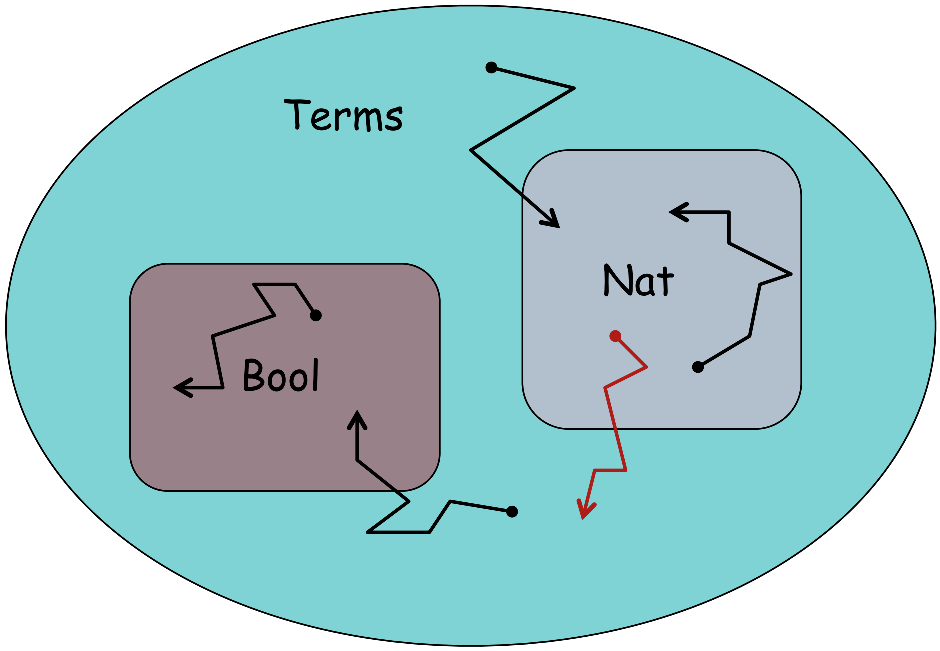
Bool and Nat values



Bool and Nat types



Evaluation preserves type



A Type System

type expressions: `T ::= . . .`

typing relation: `t : T`

typing rules giving an inductive definition of `t:T`

The typing relation `t:T` for arithmetic expressions is the smallest binary relation between terms and types satisfying the given rules.

A term `t` is typable (or well typed) if there is some `T` such that `t:T`.

Syntax: F_1

Terms	t	$::=$			terms:
		$\|$	$\mathbf{true}$		constant true
		$\|$	$\mathbf{false}$		constant false
		$\|$	$\mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3$		conditional
		$\|$	$\mathbf{x}$		variable
		$\|$	$\lambda\,\mathbf{x} : T\,.\,t$		abstraction
		$\|$	$t_1\ t_2$		application
		$\|$	$[\,\mathbf{x} \mapsto v\,]\,t$	M	
		$\|$	nv		integer
		$\|$	$t + t'$		addition
		$\|$	$(\,t\,)$	M	
	v	$::=$			values:
		$\|$	$\mathbf{true}$		true value
		$\|$	$\mathbf{false}$		false value
		$\|$	$\lambda\,\mathbf{x} : T\,.\,t$		abstraction value
		$\|$	nv		integer value

Syntax: F_1

Types

$\rightarrow$ is a function type constructor and associates to the right
Formal arguments to functions have typing annotations.

$T,\ S,\ U$	$::=$			types:
	$\|$	$T \rightarrow T'$		type of functions
	$\|$	$(\,T\,)$	M	
	$\|$	$\mathbf{Bool}$		type of booleans
	$\|$	$\mathbf{Int}$		type of integers

Static Semantics

The typing judgment $\boxed{\Gamma \vdash t : T}$

- We need to supply a context (Γ) giving types for free variables

Typing rules

$$\frac{\mathbf{x} : T \in \Gamma}{\Gamma \vdash \mathbf{x} : T} \quad \text{T\_VAR}$$

$$\frac{\Gamma, \mathbf{x} : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda \mathbf{x} : T_1 . t_2 : T_1 \to T_2} \quad \text{T\_ABS}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\, t_2 : T_{12}} \quad \text{T\_APP}$$

Static Semantics (cont)

More typing rules

$$\frac{}{\Gamma \vdash \mathbf{true} : \mathbf{Bool}} \quad \text{T\_TRUE}$$

$$\frac{}{\Gamma \vdash \mathbf{false} : \mathbf{Bool}} \quad \text{T\_FALSE}$$

$$\frac{\Gamma \vdash t_1 : \mathbf{Bool} \qquad \Gamma \vdash t_2 : T \qquad \Gamma \vdash t_3 : T}{\Gamma \vdash \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 : T} \quad \text{T\_IF}$$

$$\frac{}{\Gamma \vdash nv : \mathbf{Int}} \quad \text{T\_VAL}$$

$$\frac{\Gamma \vdash t_1 : \mathbf{Int} \qquad \Gamma \vdash t_2 : \mathbf{Int}}{\Gamma \vdash t_1 + t_2 : \mathbf{Int}} \quad \text{T\_ADD}$$

Typing Derivations

A type derivation is a tree of instances of typing rules with the desired typing as the root.

$$\dfrac{\texttt{true : Bool} \qquad \texttt{0: Int} \qquad \dfrac{\begin{array}{c}\texttt{1: Int}\\ \texttt{0: Int}\end{array}}{\texttt{1+0: Int}}}{\texttt{if true then 0 else 1+0 : Int}}\ \text{(T-If)}$$

The shape of the derivation tree exactly matches the shape of the term being typed.

Typing Derivations in F_1

Consider the term:

`λx:int. λb:bool. if b then f x else x`

in a typing environment that maps `f` to `int→int`

$$\dfrac{\dfrac{\dfrac{\Gamma\vdash\texttt{b:Bool} \qquad \dfrac{\Gamma\vdash\texttt{f:Int}\to\texttt{Int} \quad \Gamma\vdash\texttt{x:Int}}{\Gamma\vdash\texttt{f x:Int}} \qquad \Gamma\vdash\texttt{x:Int}}{\texttt{f:Int}\to\texttt{Int,x:Int,b:Bool}\vdash\texttt{if b then f x else x : int}}}{\texttt{f:Int}\to\texttt{Int,x:Int}\vdash\lambda\texttt{b:bool.if b then f x else x :Bool}\to\texttt{Int}}}{\texttt{f:Int}\to\texttt{Int}\vdash\lambda\texttt{x:Int.}\lambda\texttt{b:Bool.if b then f x else x:Int}\to\texttt{Bool}\to\texttt{Int}}$$

where Γ = `f:Int →Int,x:Int,b:Bool`

Type Checking

Syntax-directed

- Derivations follow syntactic structure of the program

Compositional

- Understand types of terms as being built from types of subterms

Annotated

- All formal parameters are annotated with types
- No need to infer types (although this wouldn't be so hard)

Without annotations, expressions need not have a unique type

```
λ x. x : int → int
λ x. x : bool → bool
```

Operational Semantics of F_1

Call-by-value evaluation relation $\boxed{t \Downarrow t'}$

$$\frac{}{\lambda \mathbf{x} : T . t \Downarrow \lambda \mathbf{x} : T . t} \quad \text{E\_LAM}$$

$$\frac{t_1 \Downarrow \lambda \mathbf{x} : T . t_1' \qquad t_2 \Downarrow v \qquad [\mathbf{x} \mapsto v]\, t_1' \Downarrow v'}{t_1\, t_2 \Downarrow v'} \quad \text{E\_APP}$$

$$\frac{t_1 \Downarrow \mathbf{true} \qquad t_2 \Downarrow v}{\mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 \Downarrow v} \quad \text{E\_IFT}$$

$$\frac{t_1 \Downarrow \mathbf{false} \qquad t_3 \Downarrow v}{\mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 \Downarrow v} \quad \text{E\_IFF}$$

$$\frac{t_1 \Downarrow nv_1 \qquad t_2 \Downarrow nv_2 \qquad nv_1 + nv_2 = nv_3}{t_1 + t_2 \Downarrow nv_3} \quad \text{E\_PLUS}$$

$$\frac{}{v \Downarrow v} \quad \text{E\_VAL}$$

Type Soundness for F1

Safety = Progress + Preservation

Progress:

- *A well-typed term is not stuck -- either it is a value, or it can take a step according to the evaluation rules*

Preservation:

- *If a well-typed term makes a step of evaluation, the resulting term is also well-typed*

Theorem ("subject reduction")

- If $\Gamma \vdash \mathtt{t:T}$ and $\mathtt{t} \Downarrow \mathtt{v}$ then $\Gamma \vdash \mathtt{v:T}$
- By induction on $\mathtt{t} \Downarrow \mathtt{v}$
  - Need to address the issue of $\mathtt{[v2 \mapsto x]t1'}$
  - Could also use induction on typing derivations

Some Helper Lemmas

Inversion (of typing relation), e.g.

- If $\Gamma \vdash \mathtt{x:R}$ then $\mathtt{x:R} \in \Gamma$
- If $\Gamma \vdash \mathtt{true:R}$ then $\mathtt{R=Bool}$
- If $\Gamma \vdash \lambda\ \mathtt{x:T.t:R}$ then $\mathtt{R=T \to R'}$ for some $\mathtt{R'}$ with $\Gamma, \mathtt{x:T} \vdash \mathtt{t:R}$

Uniqueness (of types): a term has at most one type

Canonical forms, e.g.

- If $\mathtt{v}$ is a value of type $\mathtt{Bool}$, then $\mathtt{v}$ is either $\mathtt{true}$ or $\mathtt{false}$
- If $\mathtt{v}$ is a value of type $\mathtt{T \to R}$ then $\mathtt{v} = \lambda \mathtt{x:T.t}$

Permutation: can permute order of free var defs in Γ

Weakening: if $\Gamma \vdash \mathtt{t:T}$ and $\mathtt{x} \notin \mathtt{dom}(\Gamma)$ then $\Gamma, \mathtt{x:\ S} \vdash \mathtt{t:\ T}$

Soundness

Consider the case:

$$\frac{t_1 \Downarrow \lambda\ x{:}\tau_2 . t_1' \qquad t_2 \Downarrow v_2 \qquad [v_2/x]t_1' \Downarrow v}{t_2 \ \Downarrow\ v}$$

By derivation of $t_1\ t_2\ :\ \tau$ we know

$$\frac{\Gamma \vdash t_1 : \tau_2 \to \tau \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash\ t_1\ t_2\ :\ \tau}$$

From IH on $t_1 \Downarrow$ ..., we have $\Gamma, x{:}\tau_2 \vdash t_1' : \tau$

From IH on $t_2 \Downarrow$ ... we have $\Gamma \vdash v_2 : \tau_2$

Need to infer that $\Gamma \vdash [v_2 \mapsto x]t_1' : \tau$ and use the IH

Need a substitution lemma

Substitution Lemma

If $\Gamma, x{:}\tau\ \vdash\ t : \tau'$ and $\Gamma \vdash\ y : \tau$, then

$\Gamma \vdash [y \mapsto x]t : \tau'$

Proof: by induction on the derivation of $\Gamma, x{:}\tau\ \vdash\ t : \tau'$
(see page 106 and 107 of the text)

Significance of Type Soundness

The theorem says that the result of an evaluation has the same type as the initial expression

The theorem does not say that:

- evaluation never gets stuck (it is not a progress theorem)
- evaluation terminates

Contextual semantics

Define redexes and contexts

$$
\begin{array}{lcll}
E & ::= & & \text{contexts} \\
 & | & [\,] & \\
 & | & E + t & \\
 & | & nv + E & \\
 & | & \mathbf{if}\ E\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 & \\
 & | & E\ t & \\
 & | & (\lambda \mathbf{x} : T \,.\, t)\ E & \\
\end{array}
$$

$$
\begin{array}{lcll}
r & ::= & & \text{redexes} \\
 & | & v + v' & \\
 & | & \mathbf{if}\ v\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 & \\
 & | & (\lambda \mathbf{x} : T \,.\, t)\ v & \\
\end{array}
$$

Contextual semantics

Global reduction rule: `E[r] → E[t] iff r → t`

Local rules:

$\boxed{t \longrightarrow t'}$ Evaluation

$$\frac{}{(\lambda \mathbf{x} : T . t)\, v \longrightarrow [\mathbf{x} \mapsto v]\, t} \quad \text{E\_APP}$$

$$\frac{}{\textbf{if true then } t_1 \textbf{ else } t_2 \longrightarrow t_1} \quad \text{E\_IFT}$$

$$\frac{}{\textbf{if false then } t_1 \textbf{ else } t_2 \longrightarrow t_2} \quad \text{E\_IFF}$$

$$\frac{nv_1 + nv_2 = nv_3}{nv_1 + nv_2 \longrightarrow nv_3} \quad \text{E\_PLUS}$$

Contextual Semantics

Decomposition Lemma:

- If $\Gamma \vdash$ `t:τ` and `e` is not a value then there exists a unique `E` and r such that `t = E[r]`
  - Any well-typed term can be decomposed
  - Any well-typed non-value can make progress
- Furthermore, there exists τ' such that $\Gamma \vdash$ `r:τ'`
  - All redexes are well-typed
- Furthermore, there exists `t`' such that `r→t'` and $\Gamma \vdash$ `t':τ'`
  - Local reductions are type-preserving
- Furthermore, for any `t'`, $\Gamma \vdash$`t':τ'` implies $\Gamma \vdash$`E[t']:τ`
  - An expression keeps its type if we replace a redex by an expression of the same type

Contextual Semantics

Type preservation theorem:

- If $\Gamma \vdash$ `t:τ` and `t` $\rightarrow$ `t'` then $\Gamma \vdash$ `t':τ`

  Follows from the decomposition lemma

Progress theorem:

- If $\Gamma \vdash$ `t:τ` and `t` is not a value then there exists `t'` such that `e` can make progress: `t` $\rightarrow$ `t'`
- The progress theorem says that execution can make progress on well-typed expressions.
- Furthermore, because of type preservation, we know that the execution of a well-typed expression never gets stuck.

Typability

We may erase types from expressions systematically:

```
erase(x) = x
erase(t1 t2) = erase(t1) erase(t2)
erase(λ x:τ.t) = λ x. erase(t)
```

- E.g. `erase(λx:Bool.λy:Bool->Bool.y x)=λx.λy.y x`

Is an untyped expression typable (with respect to a given type environment)?

- Given `t`, does there exist `t'` and `τ` such that `erase(t')= t` and $\Gamma \vdash$ `t':τ`
- `λx.x` is typable in the empty environment

  There is an infinite collections of typings for this term

Erasure commutes with evaluation

$$\begin{tikzcd} t \arrow[r, "erase"] \arrow[d, "eval_{\lambda\to}"] & m \arrow[d, "eval_{\lambda}"] \\ t' \arrow[r, "erase"] & m' \end{tikzcd}$$

Theorem (9.5.2)

1. if `t` $\to$ `t'` in $\lambda_{\to}$ then `erase(t)` $\to$ `erase(t')` in λ.
2. if `erase(t)` $\to$ `m` in λ then there exists `t'` such that `t` $\to$ `t'` in $\lambda_{\to}$ and `erase(t') = m`.

Curry style and Church style

Curry:

define evaluation for untyped terms, then define the well-typed subset of terms and show that they don't exhibit bad "run-time" behaviors. Erase and then evaluate.

Church:

define the set of well-typed terms and give evaluation rules only for such well-typed terms.

Static Semantics for Product Types: F1x

Extend the semantics with (binary) tuples:

$$\begin{array}{llll}
\texttt{t::=} & | & \{\, t_1 \,,\, t_2 \,\} & \textit{pair} \\
 & | & t\,.\,1 & \textit{first pro} \\
 & | & t\,.\,2 & \textit{second} \\
\texttt{v::=} & | & \{\, t_i{}^{i \in 1..n} \,\} & \textit{tuple} \\
\texttt{T::=} & | & T_1 \times T_2 & \textit{product type}
\end{array}$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash \{\, t_1 \,,\, t_2 \,\} : T_1 \times T_2} \quad \text{T\_PAIR}$$

$$\frac{\Gamma \vdash t_1 : T_1 \times T_2}{\Gamma \vdash t_1\,.\,1 : T_1} \quad \text{T\_PROJ1}$$

Static Semantics for Product Types: F1x

Extend the small-step semantics :

$$\frac{}{\{\, v_1 \,,\, v_2 \,\}\,.\,1 \longrightarrow v_1} \quad \text{E\_PAIRBETA1}$$

$$\frac{}{\{\, v_1 \,,\, v_2 \,\}\,.\,2 \longrightarrow v_2} \quad \text{E\_PAIRBETA2}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\,.\,1 \longrightarrow t_1'\,.\,1} \quad \text{E\_PROJ1}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\,.\,2 \longrightarrow t_1'\,.\,2} \quad \text{E\_PROJ2}$$

$$\frac{t_1 \longrightarrow t_1'}{\{\, t_1 \,,\, t_2 \,\} \longrightarrow \{\, t_1' \,,\, t_2 \,\}} \quad \text{E\_PAIR1}$$

$$\frac{t_2 \longrightarrow t_2'}{\{\, v_1 \,,\, t_2 \,\} \longrightarrow \{\, v_1 \,,\, t_2' \,\}} \quad \text{E\_PAIR2}$$

Records

Records are like tuples with labels

$t ::= \ldots \mid \{ L_1 = t_1, \ldots, L_n = t_n \} \mid t.L$

New form of values:

$v ::= \{ L_1 = v_1, \ldots, L_n = v_n \}$

New form of types:

$\tau ::= \ldots \mid \{ L_1 : \tau_1, \ldots, L_n : \tau_n \}$

Follows the same basic structure as Tuples

Sum Types (System F1+)

Consider types of the form:

- either an `Int` or a `Bool`
- either a function or an `Int`

These types are called *disjoint union types*

Introduce new form of expressions and types:

$t ::= \ldots \mid \mathtt{inl}\ t \mid \mathtt{inr}\ t$
$\quad \mid \mathtt{case}\ t\ \mathtt{of}\ \mathtt{inl}\ x \Rightarrow t_1 \mid \mathtt{inr}\ y \Rightarrow t_2$

$\tau ::= \ldots \mid \tau_1 + \tau_2$

- A value of type $\tau_1+\tau_2$ is either a τ_1 or a τ_2
- Similar to unions in C or Pascal, but safe
  Distinguishing between components is under compiler control
- `case` is a binding operator: `x` bound in t_1 and `y` bound in t_2

Examples

Consider the type "`unit`" with a single element called "`*`"

The type optional integer defined as "`unit + int`"

- Similar to option types in ML
- No argument: `inl *`
- Argument is 5: `inr 5`

To use the argument, must test its kind:

```
case arg of
        inl * ⇒ 0
      | inr y ⇒ y + 3
```

`inl`/`inr` are value-carrying tags and `case` does tag checking

Sum Types

Can think of `bool` as a unit type:

```
bool = unit + unit
true = inl *
false = inr *
```

```
if t then t1 else t2
```

is the same as

```
case t of
    inl * ⇒ t1
  | inr * ⇒ t2
```

Typing Rules for Sum Types

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \mathsf{inl}\, t_1 : T_1 + T_2} \quad \text{T\_INL}$$

$$\frac{\Gamma \vdash t_1 : T_2}{\Gamma \vdash \mathsf{inr}\, t_1 : T_1 + T_2} \quad \text{T\_INR}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_0 : T_1 + T_2 \\ \Gamma, x_1 : T_1 \vdash t_1 : T \\ \Gamma, x_2 : T_2 \vdash t_2 : T\end{array}}{\Gamma \vdash \mathsf{case}\, t_0 \,\mathsf{of}\, \mathsf{inl}\, x_1 \Rightarrow t_1 \mid \mathsf{inr}\, x_2 \Rightarrow t_2 : T} \quad \text{T\_CASESM}$$

Dynamic Semantics

$$\frac{}{\mathsf{case}\,(\,\mathsf{inl}\, v_0\,)\,\mathsf{of}\,\mathsf{inl}\, x_1 \Rightarrow t_1 \mid \mathsf{inr}\, x_2 \Rightarrow t_2 \longrightarrow [\,x_1 \mapsto v_0\,]\, t_1} \quad \text{E\_CASEINL}$$

$$\frac{}{\mathsf{case}\,(\,\mathsf{inr}\, v_0\,)\,\mathsf{of}\,\mathsf{inl}\, x_1 \Rightarrow t_1 \mid \mathsf{inr}\, x_2 \Rightarrow t_2 \longrightarrow [\,x_2 \mapsto v_0\,]\, t_2} \quad \text{E\_CASEINR}$$

$$\frac{t_0 \longrightarrow t_0'}{\mathsf{case}\, t_0 \,\mathsf{of}\,\mathsf{inl}\, x_1 \Rightarrow t_1 \mid \mathsf{inr}\, x_2 \Rightarrow t_2 \longrightarrow \mathsf{case}\, t_0' \,\mathsf{of}\,\mathsf{inl}\, x_1 \Rightarrow t_1 \mid \mathsf{inr}\, x_2 \Rightarrow t_2} \quad \text{E\_CASESM}$$

$$\frac{t_1 \longrightarrow t_1'}{\mathsf{inl}\, t_1 \longrightarrow \mathsf{inl}\, t_1'} \quad \text{E\_INL}$$

$$\frac{t_1 \longrightarrow t_1'}{\mathsf{inr}\, t_1 \longrightarrow \mathsf{inr}\, t_1'} \quad \text{E\_INR}$$

Type Soundness for F1+

- Type soundness still holds
- There is no way to use a value of $\tau_1 + \tau_2$ inappropriately
- The key is that the only way to use a value of $\tau_1+\tau_2$ is with case which ensures that one does not interchange a τ_1 for a τ_2
- Still, something is missing
  - How to infer τ resp. $\tau 1$ or $\tau 2$
- In C or Pascal, proper tag checking is the responsibility of the programmer (unsafe)