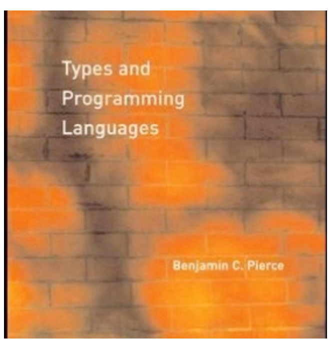
Simply-Typed λ-Calculus

Lecture 11
CS 565



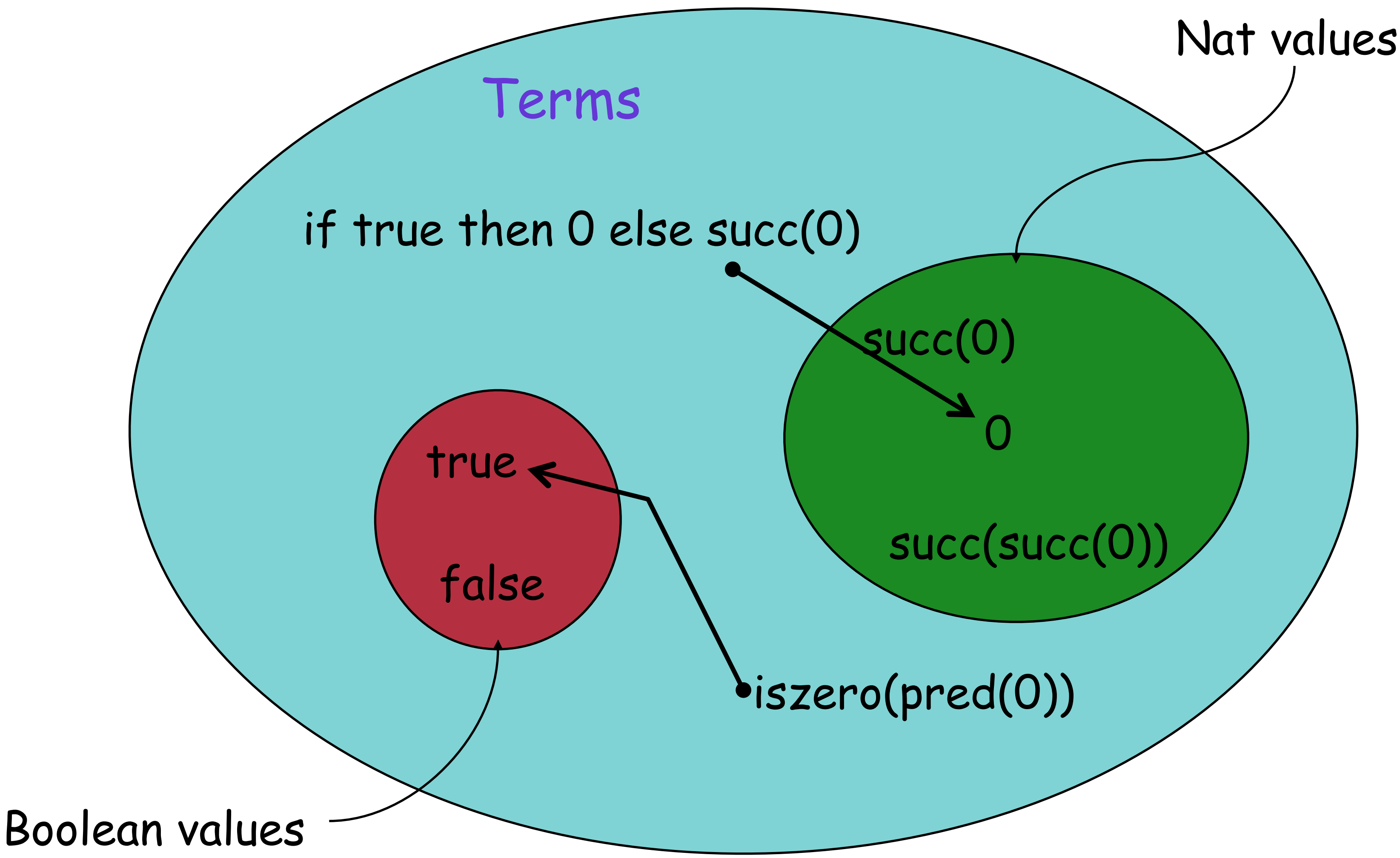
Boolean and Nat terms

Some terms represent booleans, some represent natural numbers.

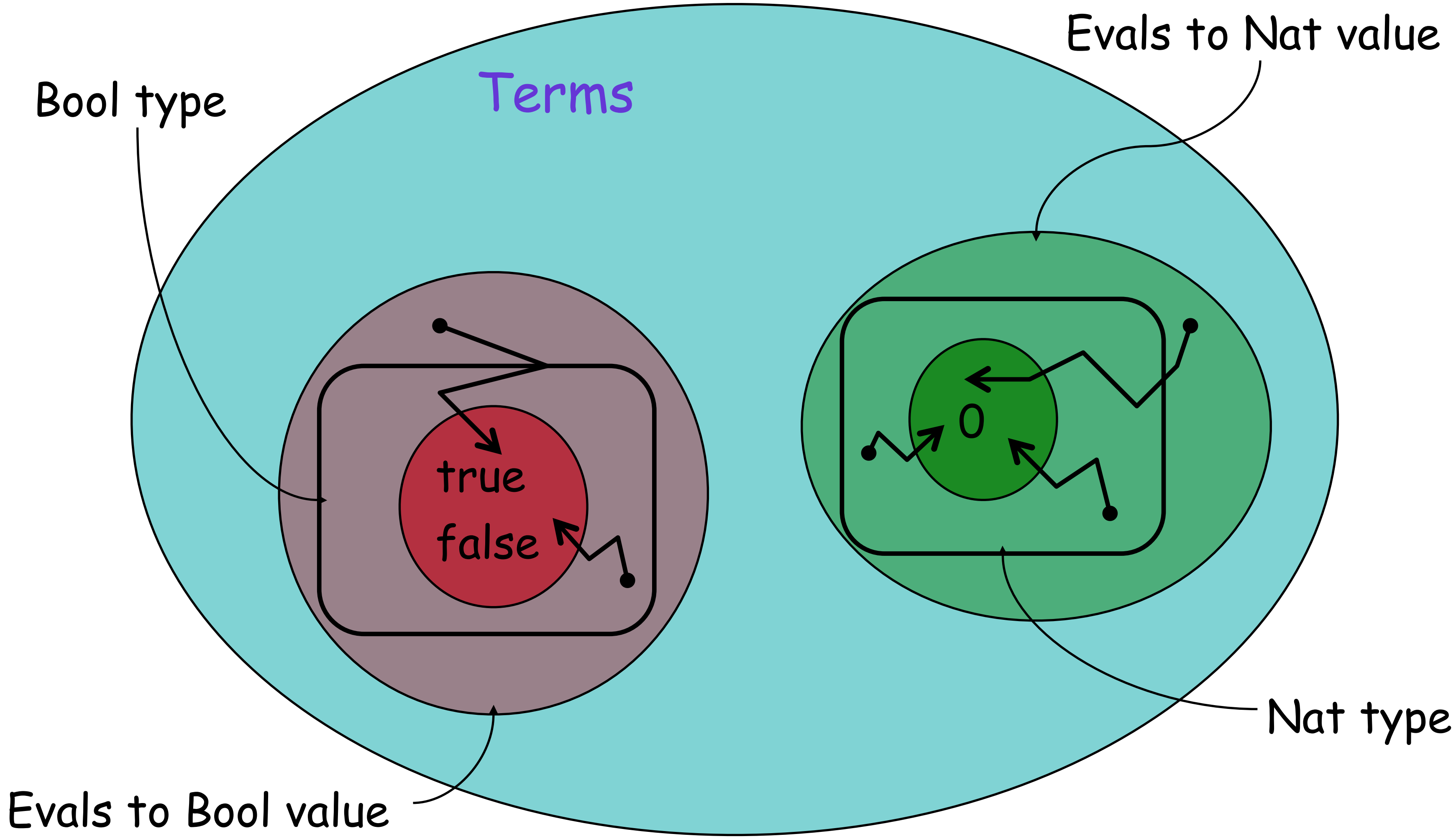
t :: = true
false
if t then t else t
0
succ t
pred t
iszero t

if t then t else t
if t then t else t

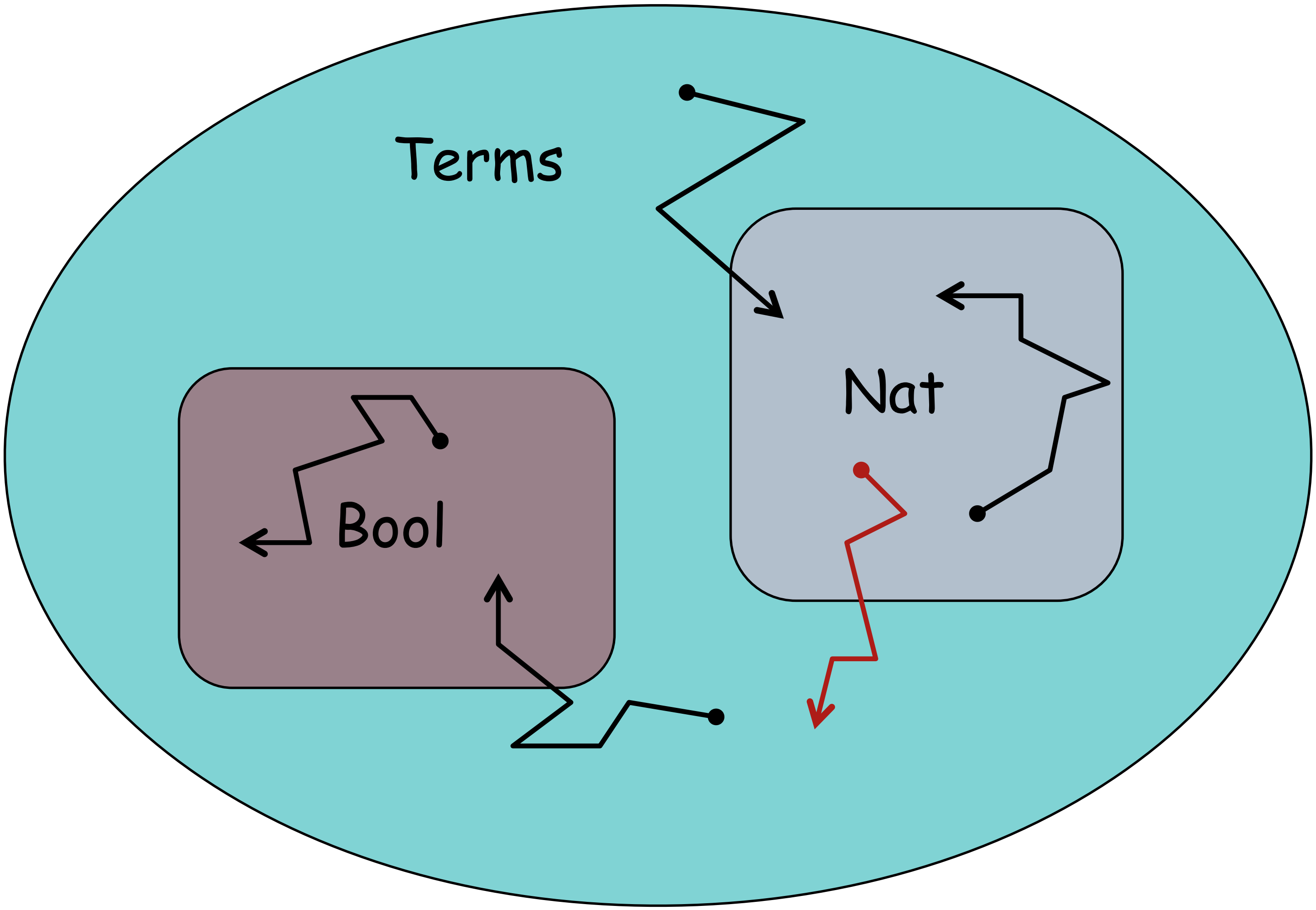
Bool and Nat values



Bool and Nat types



Evaluation preserves type



A Type System

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

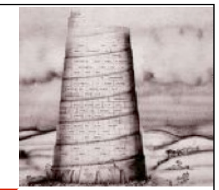
typing relation: `t : T`

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

Syntax: F_1

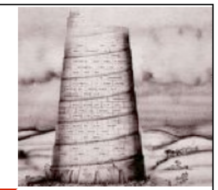
- Types

  $\rightarrow$ is a function type constructor and associates to the right

  Formal arguments to functions have typing annotations.

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

Static Semantics



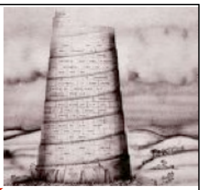
- The typing judgment $\boxed{\Gamma \vdash t : T}$
  - We need to supply a context (Γ) giving types for free variables
- Typing rules

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

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

$$\frac{\begin{array}{l}\Gamma \vdash t_1 : T_{11} \to T_{12} \\ \Gamma \vdash t_2 : T_{11}\end{array}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{T\_APP}$$

Static Semantics (cont)



- More typing rules

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

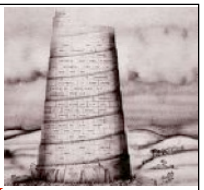
$$\frac{}{\Gamma \vdash \mathbf{false} \ : \ \mathbf{Bool}} \quad \text{T\_FALSE}$$

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

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

$$\frac{\begin{array}{c}\Gamma \vdash t_1 \ : \ \mathbf{Int} \\ \Gamma \vdash t_2 \ : \ \mathbf{Int}\end{array}}{\Gamma \vdash t_1 + t_2 \ : \ \mathbf{Int}} \quad \text{T\_ADD}$$

Typing Derivations

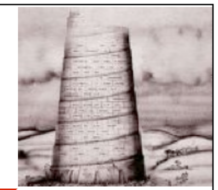


A type derivation is a tree of instances of typing rules with the desired typing as the root.

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

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

Typing Derivations in F_1



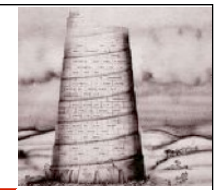
- Consider the term:

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

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

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

where Γ = $\texttt{f:Int}\rightarrow\texttt{Int,x:Int,b:Bool}$

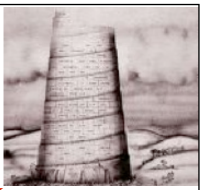


Type Checking

- 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 $\boxed{t \Downarrow t'}$

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

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

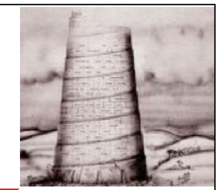
$$\frac{t_1 \Downarrow \mathbf{true} \qquad t_2 \Downarrow v}{\mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 \Downarrow v} \quad \text{E\_IFT}$$

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

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

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

Type Soundness for F1

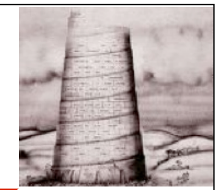


- *Safety = Progress + Preservation*
- Progress:
  - *A well-typed term is not stuck -- either it is a value, or it can take a step according to the evaluation rules*
- Preservation:
  - *If a well-typed term makes a step of evaluation, the resulting term is also well-typed*
- Theorem ("subject reduction")
  - If $\Gamma \vdash \texttt{t} : \tau$ and $\texttt{t} \Downarrow \texttt{v}$ then $\Gamma \vdash \texttt{v} : \tau$
  - By induction on $\texttt{t} \Downarrow \texttt{v}$

    Need to address the issue of $[\texttt{v2} \mapsto \texttt{x}]\texttt{t1}'$

    Could also use induction on typing derivations

Some Helper Lemmas



- Inversion (of typing relation), e.g.
  - If $\Gamma \vdash \texttt{x:R}$ then $\texttt{x:R} \in \Gamma$
  - If $\Gamma \vdash \texttt{true:R}$ then $\texttt{R=Bool}$
  - If $\Gamma \vdash \lambda\ \texttt{x:T.t:R}$ then $\texttt{R=T} \rightarrow \texttt{R}'$ for some $\texttt{R}'$ with $\Gamma,\texttt{x:T} \vdash \texttt{t:R}$
- Uniqueness (of types): a term has at most one type
- Canonical forms, e.g.
  - If $\texttt{v}$ is a value of type $\texttt{Bool}$, then $\texttt{v}$ is either $\texttt{true}$ or $\texttt{false}$
  - If $\texttt{v}$ is a value of type $\texttt{T} \rightarrow \texttt{R}$ then $\texttt{v} = \lambda\texttt{x:T.t}$
- Permutation: can permute order of free var defs in Γ
- Weakening: if $\Gamma \vdash \texttt{t:T}$ and $\texttt{x} \notin \texttt{dom}(\Gamma)$ then $\Gamma, \texttt{x: S} \vdash \texttt{t: T}$

Soundness

- Consider the case:

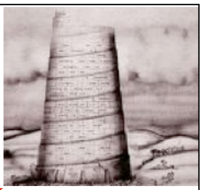
$$\frac{t_1 \Downarrow\ x{:}\tau_2.t_1' \qquad t_2 \Downarrow v_2 \qquad [x \mapsto v_2]t_1' \Downarrow v}{t_1\ t_2\ \Downarrow\ v}$$

- 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 $\Gamma \vdash [x \mapsto v_2]t_1' : \tau$ and use the IH

  Need a substitution lemma



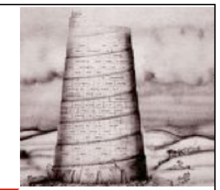
Substitution Lemma

- If Γ,$\mathtt{x:\tau} \vdash \mathtt{t:\tau'}$ and $\Gamma \vdash \mathtt{s:\tau}$, then

  $\Gamma \vdash [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t:\tau'}$

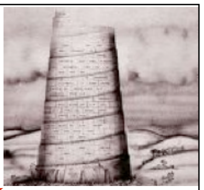
- Proof: by induction on the derivation of Γ,$\mathtt{x:\tau} \vdash \mathtt{t:\tau'}$
- (see page 106 and 107 of the text)

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

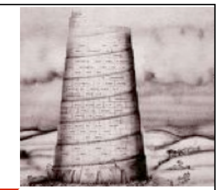
Contextual semantics



- Define redexes and contexts

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

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 `t` 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`$\rightarrow$`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 $\Gamma \vdash \mathtt{t:}\tau$ and $\mathtt{t} \rightarrow \mathtt{t'}$ then $\Gamma \vdash \mathtt{t':}\tau$

    Follows from the decomposition lemma
- Progress theorem:
  - If $\Gamma \vdash \mathtt{t:}\tau$ and $\mathtt{t}$ is not a value then there exists $\mathtt{t'}$ such that $\mathtt{e}$ can make progress: $\mathtt{t} \rightarrow \mathtt{t'}$
  - 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

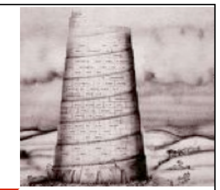
Erasure commutes with evaluation

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

Theorem (9.5.2)

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

Static Semantics for Product Types: F1x

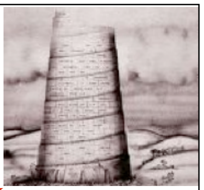


- Extend the semantics with (binary) tuples:

`t::=`	|	$\{ \mathtt{t}_1 , \mathtt{t}_2 \}$	*pair*
	|	$\mathtt{t}.1$	*first pro*
	|	$\mathtt{t}.2$	*second p*
`v::=`	|	$\{ \mathtt{t}_i{}^{i \in 1..n} \}$	*tuple*
`T::=`	|	$\mathsf{T}_1 \times \mathsf{T}_2$	*product type*

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathsf{T}_1 \qquad \Gamma \vdash \mathtt{t}_2 : \mathsf{T}_2}{\Gamma \vdash \{ \mathtt{t}_1 , \mathtt{t}_2 \} : \mathsf{T}_1 \times \mathsf{T}_2} \quad \text{T\_PAIR}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathsf{T}_1 \times \mathsf{T}_2}{\Gamma \vdash \mathtt{t}_1 .1 : \mathsf{T}_1} \quad \text{T\_PROJ1}$$



Static Semantics for Product Types: F1x

- Extend the small-step semantics :

$$\frac{}{\{\, \mathsf{v}_1 \,,\, \mathsf{v}_2 \,\}\,.\,1 \;\longrightarrow\; \mathsf{v}_1} \quad \textsc{E\_PairBeta1}$$

$$\frac{}{\{\, \mathsf{v}_1 \,,\, \mathsf{v}_2 \,\}\,.\,2 \;\longrightarrow\; \mathsf{v}_2} \quad \textsc{E\_PairBeta2}$$

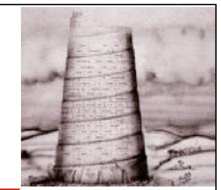
$$\frac{\mathsf{t}_1 \;\longrightarrow\; \mathsf{t}_1'}{\mathsf{t}_1\,.\,1 \;\longrightarrow\; \mathsf{t}_1'\,.\,1} \quad \textsc{E\_Proj1}$$

$$\frac{\mathsf{t}_1 \;\longrightarrow\; \mathsf{t}_1'}{\mathsf{t}_1\,.\,2 \;\longrightarrow\; \mathsf{t}_1'\,.\,2} \quad \textsc{E\_Proj2}$$

$$\frac{\mathsf{t}_1 \;\longrightarrow\; \mathsf{t}_1'}{\{\, \mathsf{t}_1 \,,\, \mathsf{t}_2 \,\} \;\longrightarrow\; \{\, \mathsf{t}_1' \,,\, \mathsf{t}_2 \,\}} \quad \textsc{E\_Pair1}$$

$$\frac{\mathsf{t}_2 \;\longrightarrow\; \mathsf{t}_2'}{\{\, \mathsf{v}_1 \,,\, \mathsf{t}_2 \,\} \;\longrightarrow\; \{\, \mathsf{v}_1 \,,\, \mathsf{t}_2' \,\}} \quad \textsc{E\_Pair2}$$

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:

$$\texttt{t ::= ...} \mid \texttt{inl t} \mid \texttt{inr t}$$
$$\mid \texttt{case t of inl } x \Rightarrow t_1 \mid \texttt{inr } y \Rightarrow t_2$$

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

  - 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

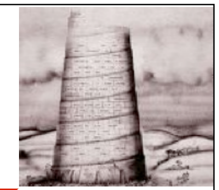
    Distinguishing between components is under compiler control
  - `case` is a binding operator: `x` bound in t_1 and `y` bound in t_2

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
```

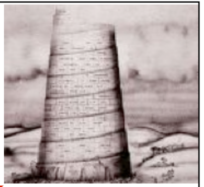
Typing Rules for Sum Types

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{T}_1}{\Gamma \vdash \mathsf{inl}\,\mathsf{t}_1 : \mathsf{T}_1 + \mathsf{T}_2} \quad \text{T\_INL}$$

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{T}_2}{\Gamma \vdash \mathsf{inr}\,\mathsf{t}_1 : \mathsf{T}_1 + \mathsf{T}_2} \quad \text{T\_INR}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathsf{t}_0 : \mathsf{T}_1 + \mathsf{T}_2 \\ \Gamma, \mathsf{x}_1 : \mathsf{T}_1 \vdash \mathsf{t}_1 : \mathsf{T} \\ \Gamma, \mathsf{x}_2 : \mathsf{T}_2 \vdash \mathsf{t}_2 : \mathsf{T}\end{array}}{\Gamma \vdash \mathsf{case}\,\mathsf{t}_0\,\mathsf{of}\,\mathsf{inl}\,\mathsf{x}_1 \Rightarrow \mathsf{t}_1 \mid \mathsf{inr}\,\mathsf{x}_2 \Rightarrow \mathsf{t}_2 : \mathsf{T}} \quad \text{T\_CASESM}$$

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
- In C or Pascal, proper tag checking is the responsibility of the programmer (unsafe)