Lesson 4
Typed Arithmetic
Typed Lambda Calculus

1/18/05

Chapters 8, 9, 10

Outline

- Types for Arithmetic
  - types
  - the typing relation
  - safety = progress + preservation
- The simply typed lambda calculus
  - function types
  - the typing relation
  - Curry-Howard correspondence
  - Erasure: Curry-style vs Church-style
- Implementation

Terms for arithmetic

Terms

t :: = true

false

if t then t else t

0

succ t

pred t

iszero t

Values

v :: = true

false

nv

nv ::= 0

succ nv

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 (boolean)
→ if t then t else t (natural number)

Nonsense terms

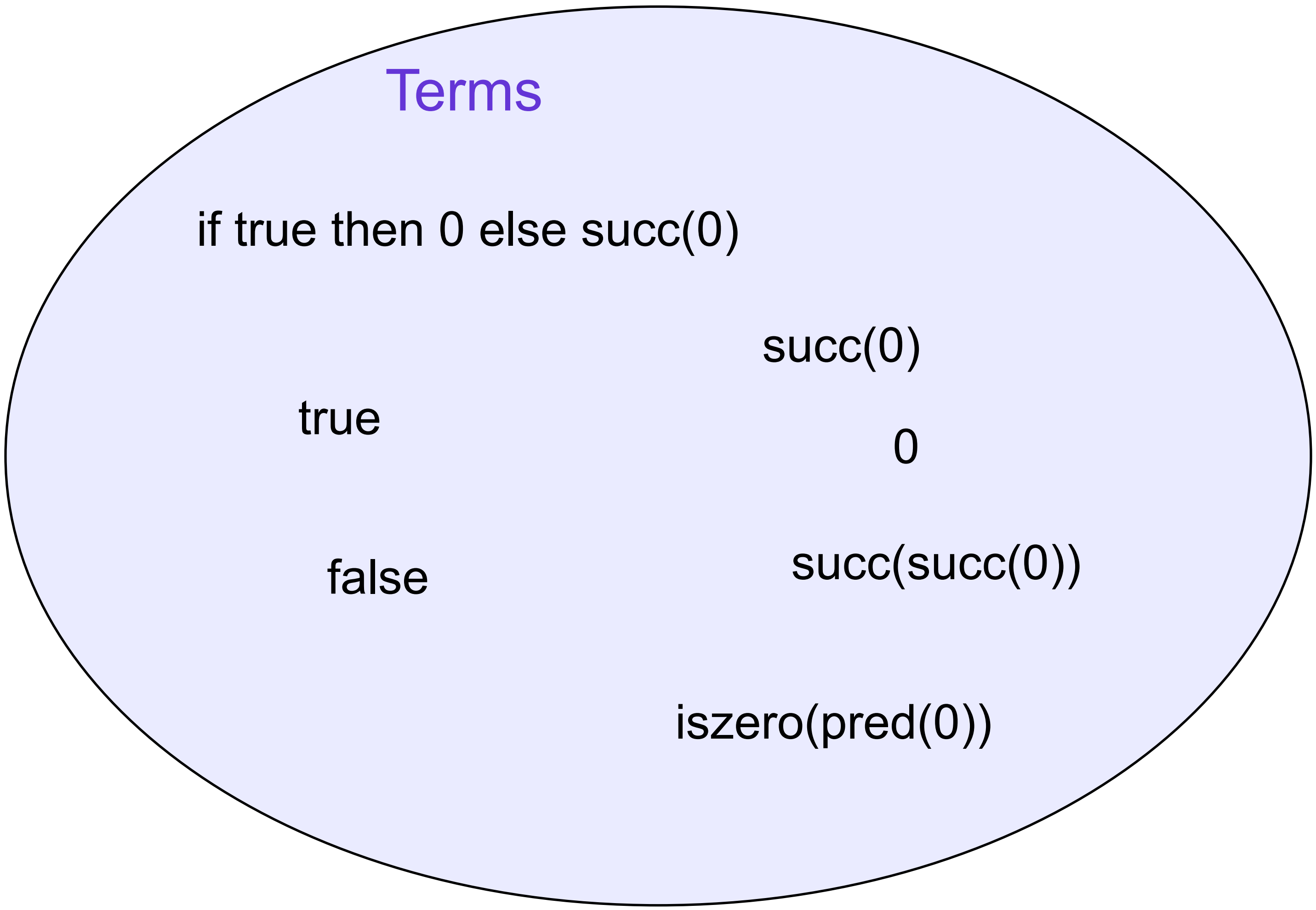
Some terms don't make sense. They represent neither booleans nor natural numbers.

succ true
iszero false
if succ(0) then true else false

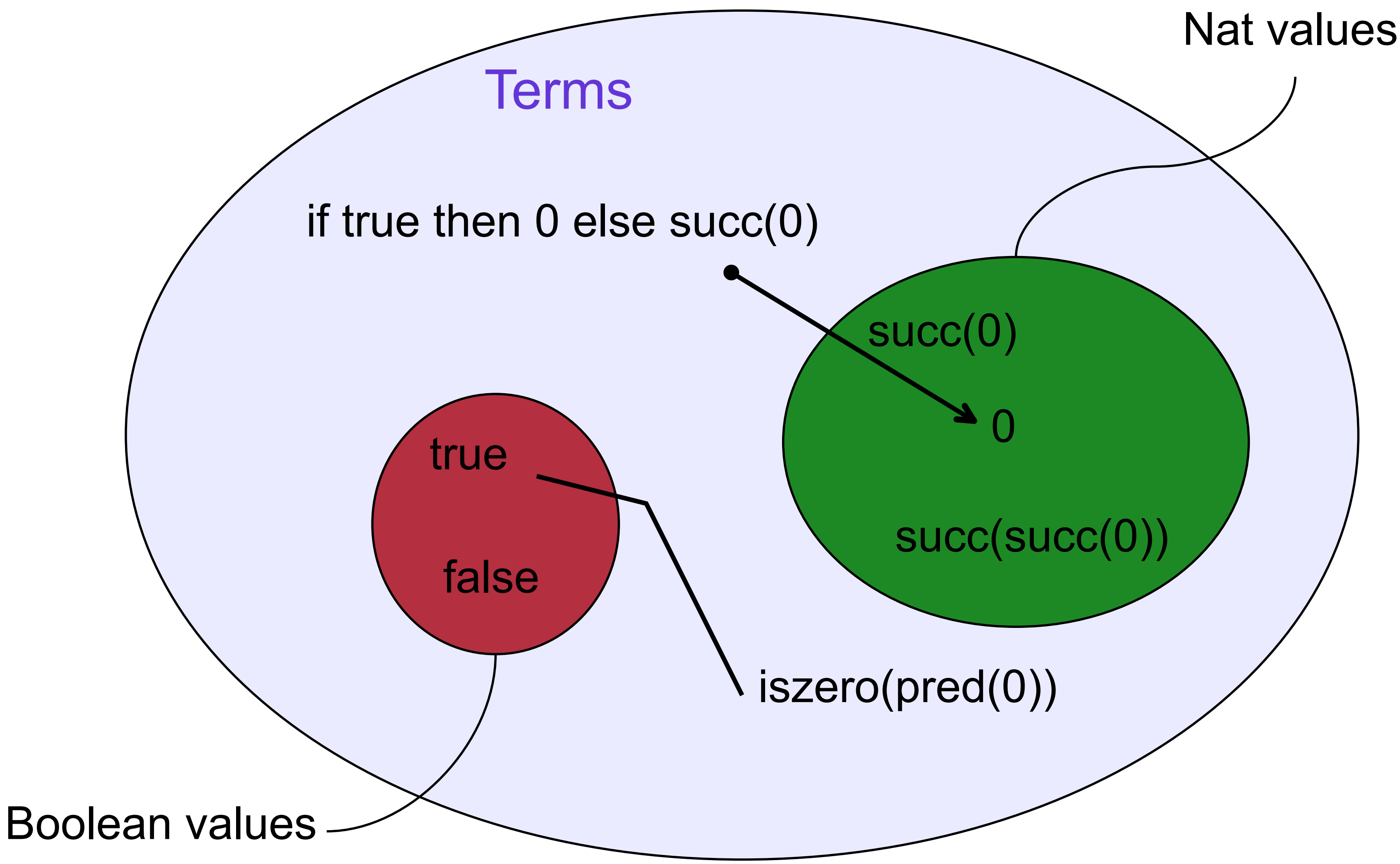
These terms are *stuck* -- no evaluation rules apply, but they are not values.
But what about the following?

if iszero(0) then true else 0

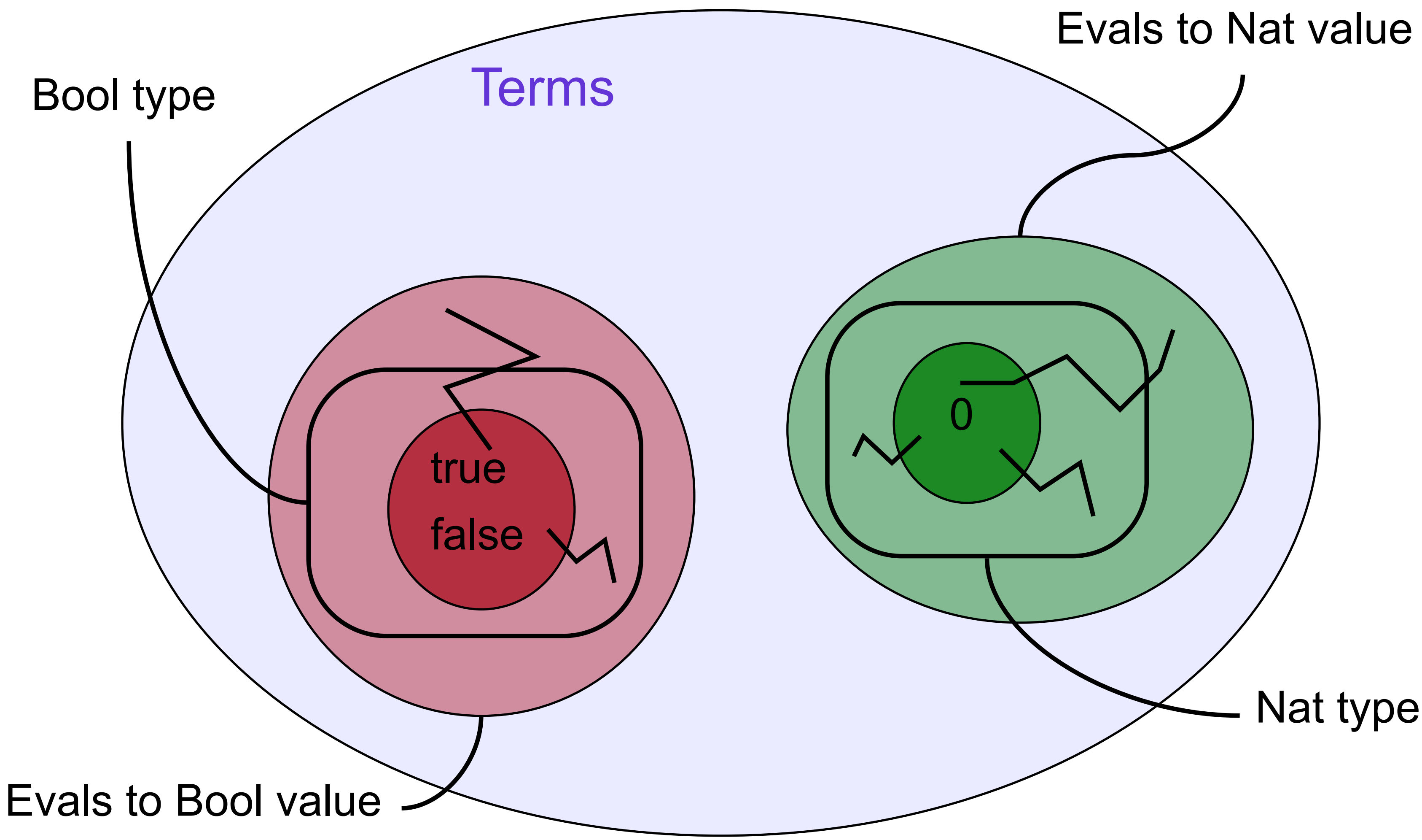
Space of terms



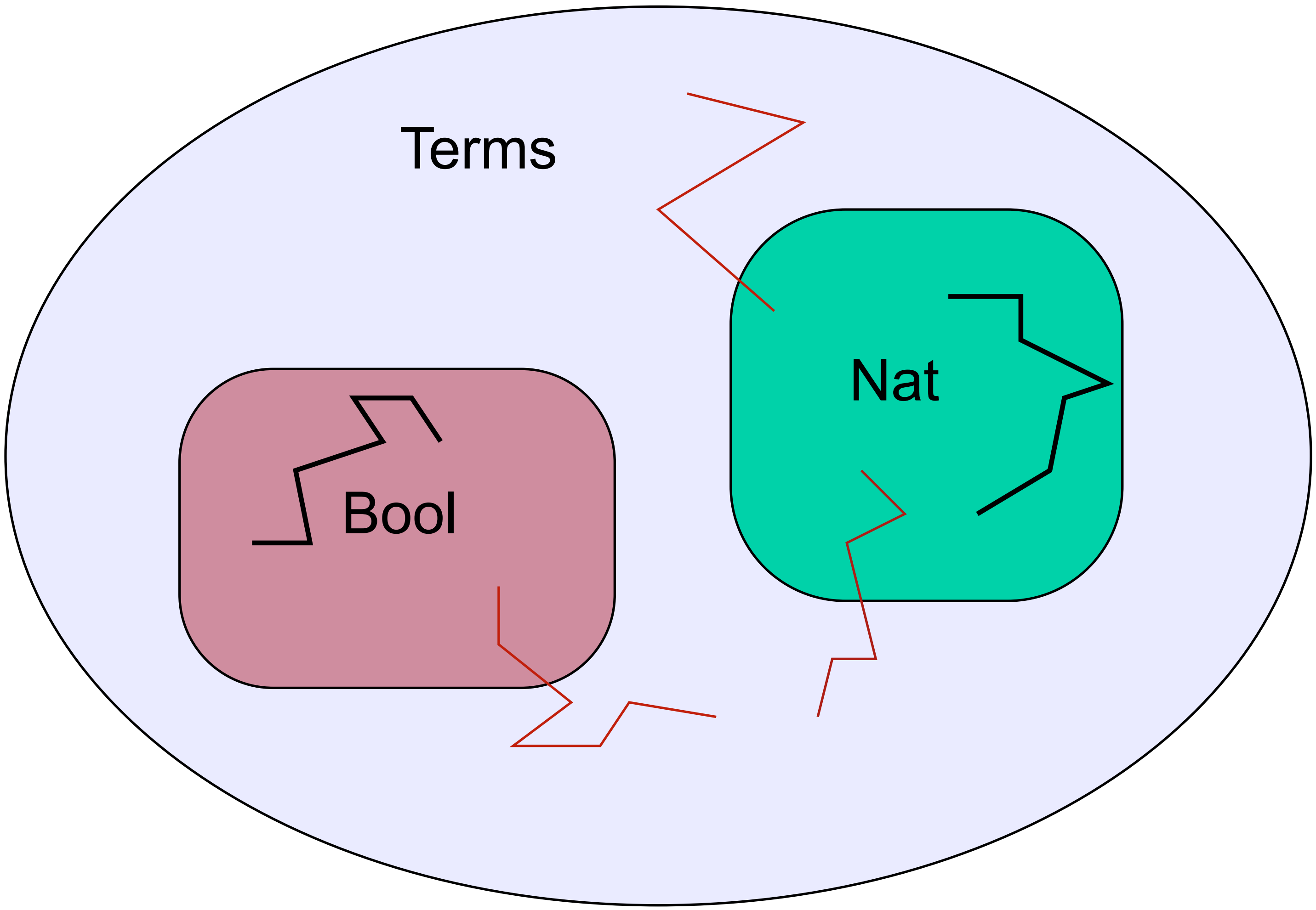
Bool and Nat values



Bool and Nat types



Evaluation preserves type



A Type System

1. type expressions: T ::= . . .
2. typing relation : t : T
3. typing rules giving an inductive definition of t : T

Typing rules for Arithmetic: **BN** (typed)

T ::= Bool | Nat (type expressions)

true : Bool (T-True)

false : Bool (T-False)

0 : Nat (T-Zero)

$$\frac{t : Nat}{succ\ t : Nat} \quad \text{(T-Succ)}$$

$$\frac{t : Nat}{pred\ t : Nat} \quad \text{(T-Pred)}$$

$$\frac{t : Nat}{iszero\ t : Bool} \quad \text{(T-IsZero)}$$

$$\frac{t_1 : Bool \qquad t_2 : T \qquad t_3 : T}{if\ t_1\ then\ t_2\ else\ t_3 : T} \quad \text{(T-If)}$$

Typing relation

Defn: 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.

Inversion Lemma

Lemma (8.2.2). [*Inversion of the typing relation*]

1. If true : R then R = Bool
2. If false : R then R = Bool
3. If if t_1 then t_2 else t_3 : R then t_1 : Bool and t_2, t_3 : R
4. If 0: R then R = Nat
5. If succ t : R then R = Nat and t : Nat
6. If pred t : R then R = Nat and t : Nat
7. If iszero t : R then R = Bool and t : Nat

Typing Derivations

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

$$
\dfrac{\dfrac{0 : \text{Nat}\ \text{(T-Zero)}}{\text{iszero}(0) : \text{Bool}}\ \text{(T-IsZero)} \qquad 0 : \text{Nat} \qquad \dfrac{0 : \text{Nat}\ \text{(T-Zero)}}{\text{pred}(0) : \text{Nat}}\ \text{(T-Pred)}}{\text{if iszero}(0)\ \text{then}\ 0\ \text{else pred}\ 0 : \text{Nat}}\ \text{(T-If)}
$$

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

Uniqueness of types

Theorem (8.2.4). Each term t has at most one type. That is, if t is typable, then its type is unique, and there is a unique derivation of its type.

Safety (or Soundness)

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.

Preservation is also known as "*subject reduction*"

Cannonical forms

Defn: a cannonical form is a well-typed value term.

Lemma (8.3.1).

1. If v is a value of type Bool, then v is true or v is false.
2. If v is a value of type Nat, then v is a numeric value, i.e. a term in nv, where

   nv ::= 0 | succ nv.

Progress and Preservation for Arithmetic

Theorem (8.3.2) [*Progress*]
If t is a well-typed term (that is, t: T for some type T), then either t is a value or else t → t' for some t'.

Theorem (8.3.3) [*Preservation*]
If t: T and t → t' then t' : T.

Proofs are by induction on the derivation of t: T.

Simply typed lambda calculus

To type terms of the lambda calculus, we need types for functions (lambda terms):

$T1 \rightarrow T2$

A function type $T1 \rightarrow T2$ specifies the argument type T1 and the result type T2 of the function.

Simply typed lambda calculus

The abstract syntax of type terms is

T ::= base types
  T → T

We need base types (e.g Bool) because otherwise we could build no type terms.

We also need terms of these base types,so we have an “applied” lambda calculus. In this case, we will take Bool as the sole base type and add corresponding Boolean terms.

Abstract syntax and values

Terms

t :: = true
  false
  if t then t else t
  x
  λx: T . t
  t t

Values

v :: = true
  false
  λx: T . t

Note that terms contain types! Lambda expressions are explicitly typed.

Typing rule for lambda terms

$$\frac{\Gamma, x{:}\ T1 \vdash t2 : T2}{\Gamma \vdash \lambda x{:}\ T1.\ t2 : T1 \rightarrow T2} \quad \text{(T-Abs)}$$

The body of a lambda term (usually) contains free variable occurrences. We need to supply a context (Γ) that gives types for the free variables.

Defn: A typing context Γ is a list of free variables with their types. A variable can appear only once in a context.

$$\Gamma ::= \varnothing \mid \Gamma, x{:}\ T$$

Typing rule for applications

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \qquad \text{(T-App)}$$

The type of the argument term must agree with the argument type of the function term.

Typing rule for variables

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-Var)}$$

The type of a variable is taken from the supplied context.

Inversion of typing relation

Lemma (9.3.1). [*Inversion of the typing relation*]

1. If $\Gamma \vdash x : R$ then $x: R \in \Gamma$
2. If $\Gamma \vdash \lambda x: T1.\ t2 : R$ then $R = T1 \rightarrow R2$ for some R2 with $\Gamma, x: T1 \vdash t2 : R2$.
3. If $\Gamma \vdash t1\ t2 : R$, then there is a T11 such that $\Gamma \vdash t1: T11 \rightarrow R$ and $\Gamma \vdash t2 : T11$.
4. If $\Gamma \vdash \text{true} : R$ then R = Bool
5. If $\Gamma \vdash \text{false} : R$ then R = Bool
6. If $\Gamma \vdash \text{if } t1 \text{ then } t2 \text{ else } t3 : R$ then $\Gamma \vdash t1 : \text{Bool}$ and $\Gamma \vdash t2, t3 : R$

Uniqueness of types

Theorem (9.3.3): In a given typing context Γ containing all the free variables of term t, there is at most one type T such that $\Gamma \vdash t: T$.

Canonical Forms ($\lambda \rightarrow$)

Lemma (9.3.4):

1. If v is a value of type Bool, then v is either true or false.
2. If v is a value of type $T1 \rightarrow T2$, then v = λx: T1.t.

Progress ($\lambda\to$)

Theorem (9.3.5): Suppose t is a closed, well-typed term (so $\vdash$ t: T for some T). Then either t is a value, or t $\to$ t' for some t'.

Proof: by induction on the derivation of $\vdash$ t: T.

Note: if t is not closed, e.g. f true, then it may be in normal form yet not be a value.

Permutation and Weakening

Lemma (9.3.6) [*Permutation*]: If $\Gamma \vdash t: T$ and Δ is a permutation of Γ, then $\Delta \vdash t: T$.

Lemma (9.3.7) [*Weakening*]: If $\Gamma \vdash t: T$ and $x \notin dom(\Gamma)$, then for any type S, $\Gamma, x: S \vdash t: T$, with a derivation of the same depth.

Proof: by induction on the derivation of $\vdash t: T$.

Substitution Lemma

Lemma (9.3.8) [*Preservation of types under substitutions*]:
If $\Gamma, x\colon S \vdash t : T$ and $\Gamma \vdash s\colon S$, then $\Gamma \vdash [x \mapsto s]t\colon T$.

Proof: induction of the derivation of $\Gamma, x\colon S \vdash t : T$.
Replace leaf nodes for occurences of x with copies of the derivation of $\Gamma \vdash s\colon S$.

Preservation ($\lambda\rightarrow$)

Theorem (9.3.9) [*Preservation*]:
If $\Gamma \vdash t : T$ and $t \rightarrow t'$, then $\Gamma \vdash t' : T$.

Proof: induction of the derivation of $\Gamma \vdash t : T$, similar to the proof for typed arithmetic, but requiring the Substitution Lemma for the beta redex case.

Homework: write a detailed proof of Thm 9.3.9.

Introduction and Elimination rules

λ Introduction

$$\frac{\Gamma, x{:}\ T1 \vdash t2 : T2}{\Gamma \vdash \lambda x{:}\ T1.\ t2 : T1 \to T2} \quad \text{(T-Abs)}$$

λ Elimination

$$\frac{\Gamma \vdash t1 : T11 \to T12 \qquad \Gamma \vdash t2 : T11}{\Gamma \vdash t1\ t2 : T12} \quad \text{(T-App)}$$

Typing rules often come in *intro-elim* pairs like this.
Sometimes there are multiple intro or elim rules for a construct.

Erasure

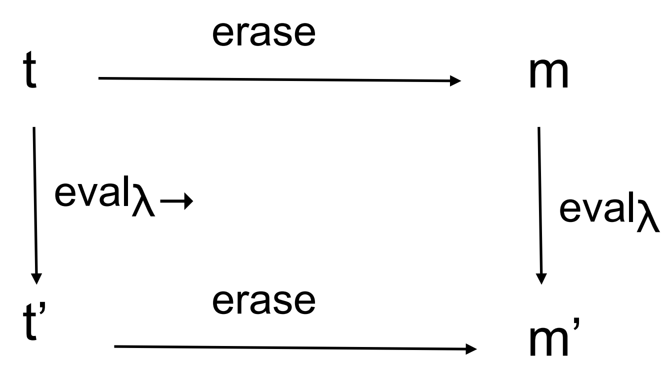
Defn: The erasure of a simply typed term is defined by:

$$\begin{aligned} \text{erase}(x) &= x \\ \text{erase}(\lambda x{:}\ T.\ t) &= \lambda x.\ \text{erase}(t) \\ \text{erase}(t1\ t2) &= (\text{erase}(t1))(\text{erase}(t2)) \end{aligned}$$

erase maps a simply typed term in $\lambda\rightarrow$ to the corresponding untyped term in λ.

$$\text{erase}(\lambda x{:}\ \text{Bool}.\ \lambda y{:}\ \text{Bool} \rightarrow \text{Bool}.\ y\ x) = \lambda x.\ \lambda y.\ y\ x$$

Erasure commutes with evaluation



Theorem (9.5.2)

1. if t → t' in λ→ then erase(t) → erase(t') in λ.
2. if erase(t) → m in λ then there exists t' such that t → t' in λ→ and erase(t') = m.

Curry style and Church style

<u>Curry</u>

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.

<u>Church</u>

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

Homework

Modify the simplebool program to add arithmetic terms and a second primitive type Nat.

1. Add Nat, 0, succ, pred, iszero tokens to lexer and parser.
2. Extend the definition of terms in the parser with arithmetic forms (see tyarith)
3. Add type and term constructors to abstract syntax in syntax.sml, and modify print functions accordingly.
4. Modify the eval and typeof functions in core.sml to handle arithmetic expressions.

Optional homework

Can you define the arithmetic plus operation in $\lambda\to$ (BN)?