Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

600 minutes

Mutually Recursive Types

```
# type 'a tree = TreeLeaf of 'a
   | TreeNode of 'a treeList
and 'a treeList = Last of 'a tree
   | More of ('a tree * 'a treeList);;
type 'a tree = TreeLeaf of 'a | TreeNode of 'a
  treeList
and 'a treeList = Last of 'a tree | More of ('a
  tree * 'a treeList)
```

Mutually Recursive Functions

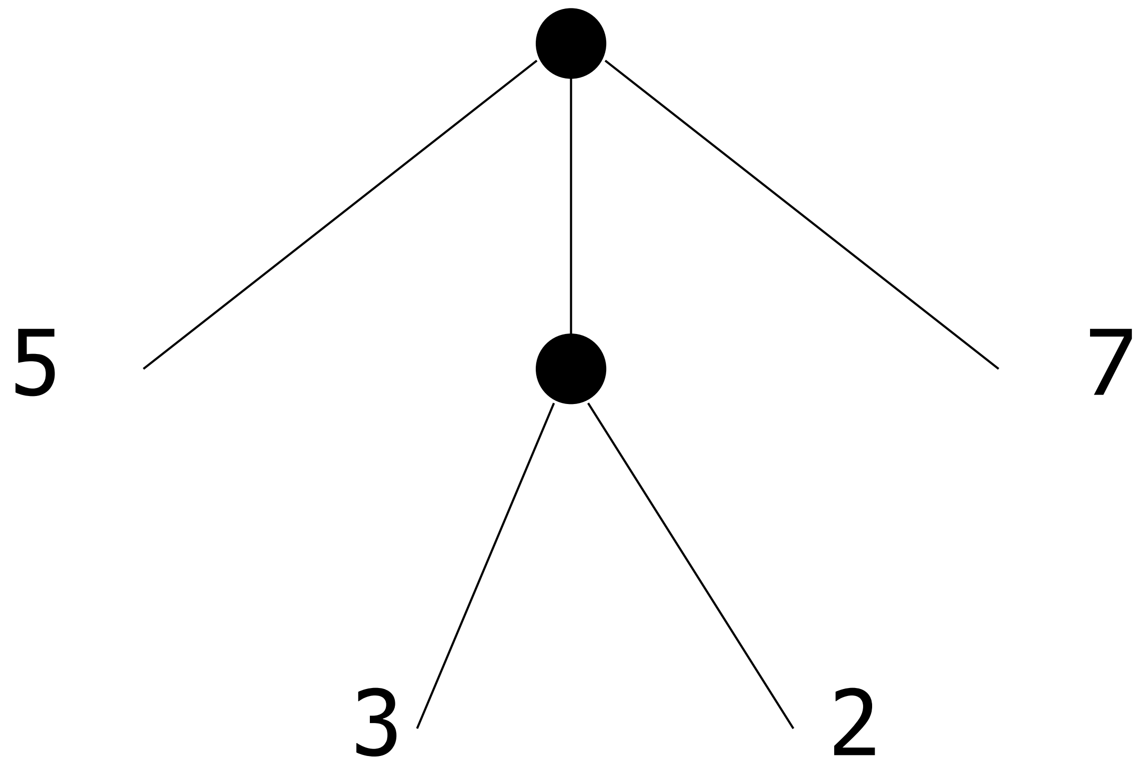
```
# let rec fringe tree =
    match tree with (TreeLeaf x) -> [x]
  | (TreeNode list) -> list_fringe list
and list_fringe tree_list =
    match tree_list with (Last tree) -> fringe tree
  | (More (tree,list)) ->
    (fringe tree) @ (list_fringe list);;

val fringe : 'a tree -> 'a list = <fun>
val list_fringe : 'a treeList -> 'a list = <fun>
```

Mutually Recursive Types - Values

```
# let tree =
   TreeNode
    (More (TreeLeaf 5,
          (More (TreeNode
                (More (TreeLeaf 3,
                      Last (TreeLeaf 2))),
                Last (TreeLeaf 7)))));;
```

Mutually Recursive Types - Values

A more conventional picture



Mutually Recursive Functions

```
# fringe tree;;
- : int list = [5; 3; 2; 7]
```

Problem

```
# type 'a tree = TreeLeaf of 'a | TreeNode of 'a treeList
and 'a treeList = Last of 'a tree | More of ('a tree * 'a treeList);;
```

Define tree_size

Problem

```
# type 'a tree = TreeLeaf of ’a | TreeNode of 'a treeList
and 'a treeList = Last of 'a tree | More of ('a tree * 'a treeList);;
```

Define tree_size

```
let rec tree_size t =
    match t with TreeLeaf _ ->
    | TreeNode ts ->
```

Problem

```
# type 'a tree = TreeLeaf of ’a | TreeNode of 'a treeList
and 'a treeList = Last of 'a tree | More of ('a tree * 'a treeList);;
```

Define tree_size

```
let rec tree_size t =
      match t with TreeLeaf _ -> 1
      | TreeNode ts -> treeList_size  ts
```

Problem

```
# type 'a tree = TreeLeaf of 'a | TreeNode of 'a treeList
and 'a treeList = Last of 'a tree | More of ('a tree * 'a treeList);;
```

Define tree_size and treeList_size

```
let rec tree_size t =
      match t with TreeLeaf _ -> 1
      | TreeNode ts -> treeList_size  ts
and treeList_size ts =
```

Problem

```
# type 'a tree = TreeLeaf of 'a | TreeNode of 'a treeList
and 'a treeList = Last of 'a tree | More of ('a tree * 'a treeList);;
```

Define tree_size and treeList_size

```
let rec tree_size t =
      match t with TreeLeaf _ -> 1
      | TreeNode ts -> treeList_size  ts
and treeList_size ts =
      match ts with Last t ->
      | More t ts' ->
```

Problem

```
# type 'a tree = TreeLeaf of 'a | TreeNode of 'a treeList
and 'a treeList = Last of 'a tree | More of ('a tree * 'a treeList);;
```

Define tree_size and treeList_size

```
let rec tree_size t =
    match t with TreeLeaf _ -> 1
    | TreeNode ts -> treeList_size  ts
and treeList_size ts =
    match ts with Last t -> tree_size t
    | More t ts' -> tree_size t + treeList_size ts'
```

Problem

```
# type 'a tree = TreeLeaf of ’a | TreeNode of 'a treeList
and 'a treeList = Last of 'a tree | More of ('a tree * 'a treeList);;
```

Define tree_size and treeList_size

```
let rec tree_size t =
      match t with TreeLeaf _ -> 1
      | TreeNode ts -> treeList_size  ts
and treeList_size ts =
      match ts with Last t -> tree_size t
      | More t ts’ -> tree_size t + treeList_size ts’
```

Nested Recursive Types

```
# type 'a labeled_tree =
 TreeNode of ('a * 'a labeled_tree
  list);;
type 'a labeled_tree = TreeNode of ('a
  * 'a labeled_tree list)
```

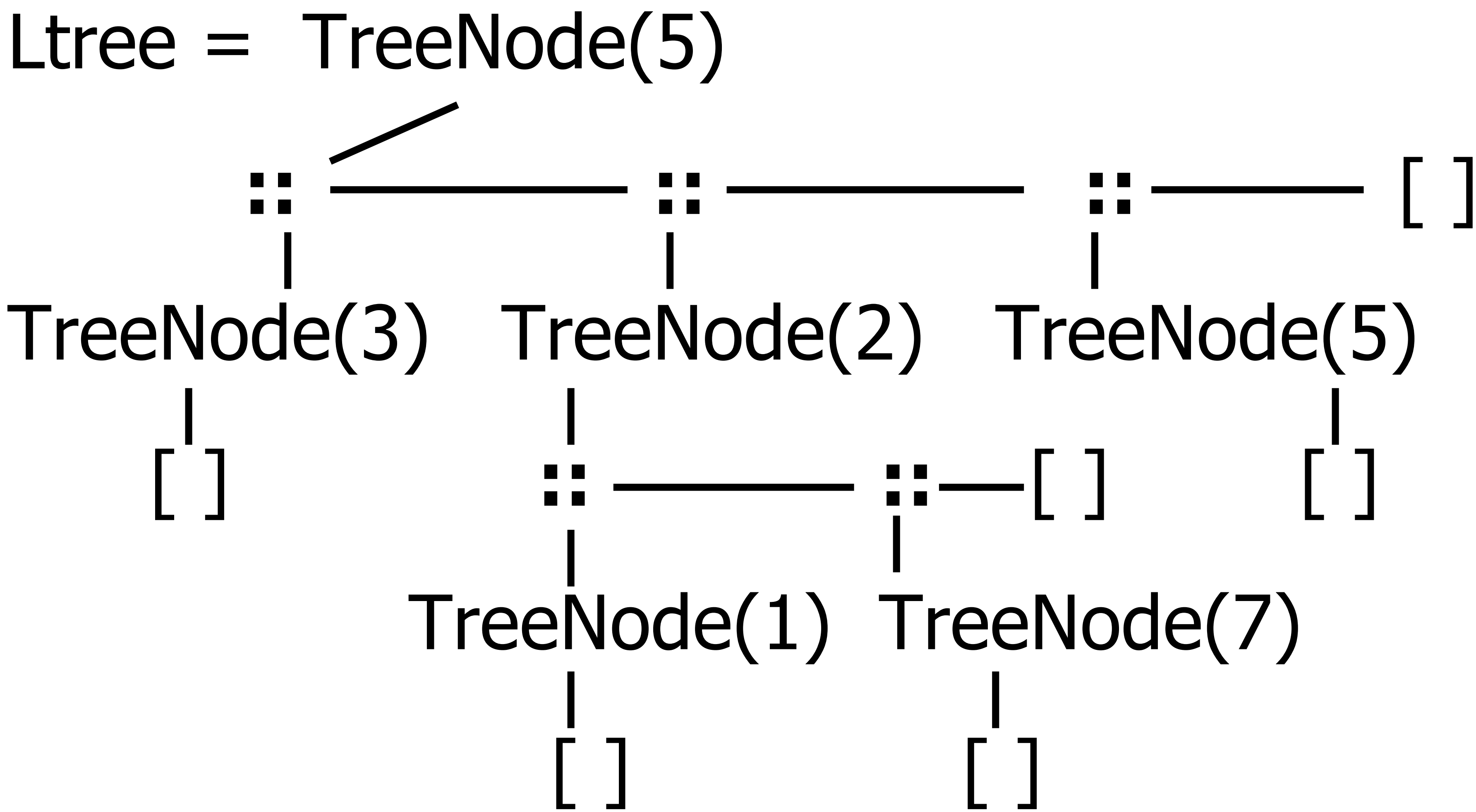
Nested Recursive Type Values

```
# let ltree =
  TreeNode(5,
    [TreeNode (3, []);
     TreeNode (2, [TreeNode (1, []);
                   TreeNode (7, [])]);
     TreeNode (5, [])]);;
```

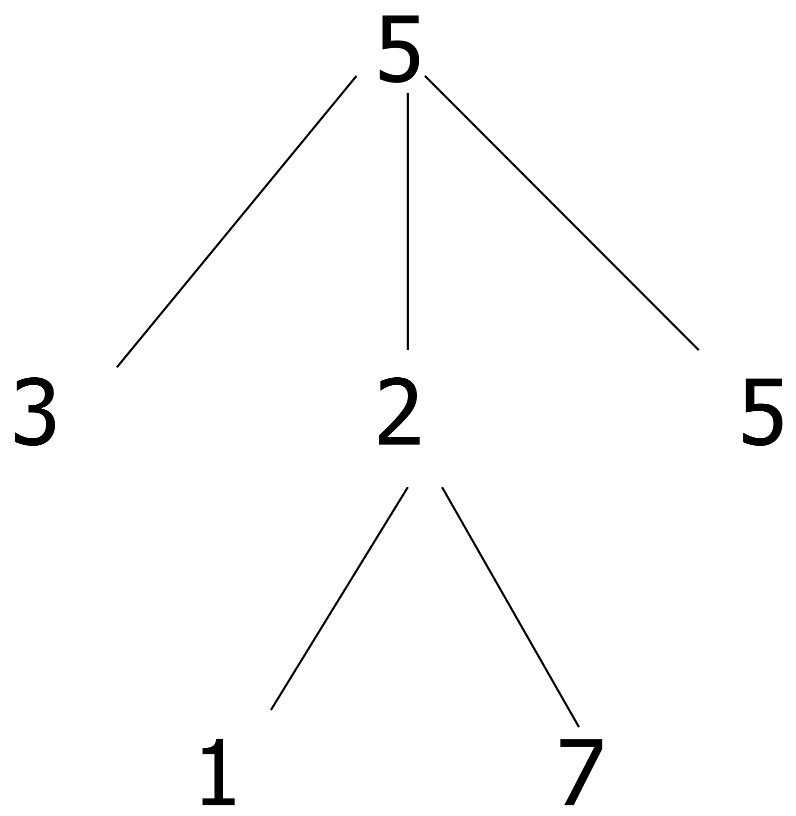
Nested Recursive Type Values

```
val ltree : int labeled_tree =
  TreeNode
   (5,
    [TreeNode (3, []); TreeNode (2,
  [TreeNode (1, []); TreeNode (7, [])]);
     TreeNode (5, [])])
```

Nested Recursive Type Values



Nested Recursive Type Values



Mutually Recursive Functions

```
# let rec flatten_tree labtree =
   match labtree with TreeNode (x,treelist)
     -> x::flatten_tree_list treelist
   and flatten_tree_list treelist =
   match treelist with [] -> []
   | labtree::labtrees
     -> flatten_tree labtree
       @ flatten_tree_list labtrees;;
```

Mutually Recursive Functions

val flatten_tree : 'a labeled_tree -> 'a list = <fun>

val flatten_tree_list : 'a labeled_tree list -> 'a list = <fun>

flatten_tree ltree;;

- : int list = [5; 3; 2; 1; 7; 5]

- Nested recursive types lead to mutually recursive functions

Why Data Types?

- Data types play a key role in:
  - *Data abstraction* in the design of programs
  - *Type checking* in the analysis of programs
  - *Compile-time code generation* in the translation and execution of programs
    - Data layout (how many words; which are data and which are pointers) dictated by type

Terminology

- Type: A type *t* defines a set of possible data values
  - E.g. short in C is $\{x \mid 2^{15} - 1 \geq x \geq -2^{15}\}$
  - A value in this set is said to have type *t*

- Type system: rules of a language assigning types to expressions

Types as Specifications

- Types describe properties
- Different type systems describe different properties, eg
  - Data is read-write versus read-only
  - Operation has authority to access data
  - Data came from “right” source
  - Operation might or could not raise an exception
- Common type systems focus on types describing same data layout and access methods

Sound Type System

- If an expression is assigned type *t*, and it evaluates to a value *v*, then *v* is in the set of values defined by *t*

- SML, OCAML, Scheme and Ada have sound type systems
- Most implementations of C and C++ do not

Strongly Typed Language

- When no application of an operator to arguments can lead to a run-time type error, language is *strongly typed*
  - Eg: 1 + 2.3;;
- Depends on definition of “type error”

Strongly Typed Language

- C++ claimed to be “strongly typed”, but
  - Union types allow creating a value at one type and using it at another
  - Type coercions may cause unexpected (undesirable) effects
  - No array bounds check (in fact, no runtime checks at all)
- SML, OCAML “strongly typed” but still must do dynamic array bounds checks, runtime type case analysis, and other checks

Static vs Dynamic Types

- *Static type*: type assigned to an expression at compile time
- *Dynamic type*: type assigned to a storage location at run time
- *Statically typed language*: static type assigned to every expression at compile time
- *Dynamically typed language*: type of an expression determined at run time

Type Checking

- When is op(arg1,…,argn) allowed?
- *Type checking* assures that operations are applied to the right number of arguments of the right types
  - Right type may mean same type as was specified, or may mean that there is a predefined implicit coercion that will be applied
- Used to resolve overloaded operations

Type Checking

- Type checking may be done *statically* at compile time or *dynamically* at run time
- Dynamically typed (aka untyped) languages (eg LISP, Prolog) do only dynamic type checking
- Statically typed languages can do most type checking statically

Dynamic Type Checking

- Performed at run-time before each operation is applied
- Types of variables and operations left unspecified until run-time
  - Same variable may be used at different types

Dynamic Type Checking

- Data object must contain type information
- Errors aren't detected until violating application is executed (maybe years after the code was written)

Static Type Checking

- Performed after parsing, before code generation
- Type of every variable and signature of every operator must be known at compile time

Static Type Checking

- Can eliminate need to store type information in data object if no dynamic type checking is needed
- Catches many programming errors at earliest point
- Can’t check types that depend on dynamically computed values
  - Eg: array bounds

Static Type Checking

- Typically places restrictions on languages
  - Garbage collection
  - References instead of pointers
  - All variables initialized when created
  - Variable only used at one type
    - Union types allow for work-arounds, but effectively introduce dynamic type checks

Type Declarations

- *Type declarations*: explicit assignment of types to variables (signatures to functions) in the code of a program
  - Must be checked in a strongly typed language
  - Often not necessary for strong typing or even static typing (depends on the type system)

Type Inference

- *Type inference*: A program analysis to assign a type to an expression from the program context of the expression
  - Fully static type inference first introduced by Robin Miller in ML
  - Haskle, OCAML, SML all use type inference
    - Records are a problem for type inference

Format of Type Judgments

- A *type judgement* has the form

$$\Gamma \text{ |- exp} : \tau$$

- Γ is a typing environment
  - Supplies the types of variables (and function names when function names are not variables)
  - Γ is a set of the form $\{ x : \sigma , \ldots \}$
  - For any x at most one σ such that $(x : \sigma \in \Gamma)$
- exp is a program expression
- τ is a type to be assigned to exp
- |- pronounced "turnstyle", or "entails" (or "satisfies" or, informally, "shows")

Axioms – Constants (Monomorphic)

$$\frac{}{\Gamma \mid\!- \; n : \text{int}}$$ (assuming *n* is an integer constant)

$$\frac{}{\Gamma \mid\!- \; \text{true} : \text{bool}} \qquad \frac{}{\Gamma \mid\!- \; \text{false} : \text{bool}}$$

- These rules are true with any typing environment
- Γ, *n* are meta-variables

Axioms – Variables (Monomorphic Rule)

Notation: Let $\Gamma(x) = \sigma$ if $x : \sigma \in \Gamma$

Note: if such σ exits, its unique

Variable axiom:

$$\frac{}{\Gamma \mid\!- \; x : \sigma} \quad \text{if } \Gamma(x) = \sigma$$

Simple Rules – Arithmetic (Mono)

Primitive Binary operators ($\oplus \in \{ +, -, *, \ldots\}$):

$$\frac{\Gamma \mid\!- \; e_1:\tau_1 \qquad \Gamma \mid\!- \; e_2:\tau_2 \qquad (\oplus):\tau_1 \rightarrow \tau_2 \rightarrow \tau_3}{\Gamma \mid\!- \; e_1 \oplus e_2 : \tau_3}$$

Special case: Relations ($\sim \in \{ <, >, =, <=, >= \}$):

$$\frac{\Gamma \mid\!- \; e_1 : \tau \qquad \Gamma \mid\!- \; e_2 : \tau \qquad (\sim):\tau \rightarrow \tau \rightarrow \text{bool}}{\Gamma \mid\!- \; e_1 \sim e_2 : \text{bool}}$$

For the moment, think τ is int

Example: {x:int} |- x + 2 = 3 :bool

What do we need to show first?

{x:int} |- x + 2 = 3 : bool

Example: {x:int} |- x + 2 = 3 :bool

What do we need for the left side?

$$\frac{\{x : int\} \mid\!- x + 2 : int \qquad \{x:int\} \mid\!- 3 : int}{\{x:int\} \mid\!- x + 2 = 3 : bool}\text{Bin}$$

Example: {x:int} |- x + 2 = 3 :bool

How to finish?

$$\dfrac{\dfrac{\{x{:}int\}\ \vert\text{-}\ x{:}int \quad \{x{:}int\}\ \vert\text{-}\ 2{:}int}{\{x : int\}\ \vert\text{-}\ x + 2 : int}\text{Bin} \qquad \{x{:}int\}\ \vert\text{-}\ 3 : int}{\{x{:}int\}\ \vert\text{-}\ x + 2 = 3 : bool}\text{Bin}$$

Example: {x:int} |- x + 2 = 3 :bool

Complete Proof (type derivation)

$$\dfrac{\dfrac{\dfrac{}{\{x:int\}\ |\!-\ x:int}\,\text{Var} \qquad \dfrac{}{\{x:int\}\ |\!-\ 2:int}\,\text{Const}}{\{x : int\}\ |\!-\ x + 2 : int}\,\text{Bin} \qquad \dfrac{}{\{x:int\}\ |\!-\ 3 :int}\,\text{Const}}{\{x:int\}\ |\!-\ x + 2 = 3 : bool}\,\text{Bin}$$

Simple Rules - Booleans

Connectives

$$\frac{\Gamma \text{ |- } e_1 : \text{bool} \qquad \Gamma \text{ |- } e_2 : \text{bool}}{\Gamma \text{ |- } e_1 \ \&\& \ e_2 : \text{bool}}$$

$$\frac{\Gamma \text{ |- } e_1 : \text{bool} \qquad \Gamma \text{ |- } e_2 : \text{bool}}{\Gamma \text{ |- } e_1 \ || \ e_2 : \text{bool}}$$

Type Variables in Rules

- If_then_else rule:

$$\frac{\Gamma \mid\!- \; e_1 : \text{bool} \quad \Gamma \mid\!- \; e_2 : \tau \quad \Gamma \mid\!- \; e_3 : \tau}{\Gamma \mid\!- \; (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) : \tau}$$

- τ is a type variable (meta-variable)
- Can take any type at all
- All instances in a rule application must get same type
- Then branch, else branch and if_then_else must all have same type

Example derivation: if-then-else-

- Γ = {x:int, int_of_float:float -> int, y:float}

$$\frac{\Gamma \mid\!- \text{(fun y -> y > 3) x : bool} \qquad \Gamma \mid\!- \text{x+2 : int} \qquad \Gamma \mid\!- \text{int\_of\_float y : int}}{\Gamma \mid\!- \text{if (fun y -> y > 3) x then x + 2 else int\_of\_float y : int}}$$

Function Application

- Application rule:

$$\frac{\Gamma \mid\!\!- \; e_1 : \tau_1 \to \tau_2 \qquad \Gamma \mid\!\!- \; e_2 : \tau_1}{\Gamma \mid\!\!- \; (e_1 \; e_2) : \tau_2}$$

- If you have a function expression e_1 of type $\tau_1 \to \tau_2$ applied to an argument e_2 of type τ_1, the resulting expression $e_1 e_2$ has type τ_2

750 minutes

Example: Application

- Γ = {x:int, int_of_float:float -> int, y:float}

Γ |- (fun y -> y > 3)
: int -> bool Γ |- x : int

Γ |- (fun y -> y > 3) x : bool

Fun Rule

- Rules describe types, but also how the environment Γ may change
- Can only do what rule allows!
- fun rule:

$$\frac{\{x : \tau_1\} + \Gamma \mid\!- \; e : \tau_2}{\Gamma \mid\!- \text{fun } x \text{ -> } e : \tau_1 \rightarrow \tau_2}$$

Fun Examples

$$\frac{\{y : int\} + \Gamma \mid\!- \; y + 3 : int}{\Gamma \mid\!- \; fun\ y \;\text{->}\; y + 3 : int \rightarrow int}$$

$$\frac{\{f : int \rightarrow bool\} + \Gamma \mid\!- \; f\ 2 :: [true] : bool\ list}{\Gamma \mid\!- \; (fun\ f \;\text{->}\; (f\ 2) :: [true]) : (int \rightarrow bool) \rightarrow bool\ list}$$

(Monomorphic) Let and Let Rec

- let rule:

$$\frac{\Gamma \mid\!- \; e_1 : \tau_1 \qquad \{x : \tau_1\} + \Gamma \mid\!- \; e_2 : \tau_2}{\Gamma \mid\!- \; (\text{let } x = e_1 \text{ in } e_2) : \tau_2}$$

- let rec rule:

$$\frac{\{x : \tau_1\} + \Gamma \mid\!- \; e_1 : \tau_1 \quad \{x : \tau_1\} + \Gamma \mid\!- \; e_2 : \tau_2}{\Gamma \mid\!- \; (\text{let rec } x = e_1 \text{ in } e_2) : \tau_2}$$

Curry - Howard Isomorphism

- Type Systems are logics; logics are type systems
- Types are propositions; propositions are types
- Terms are proofs; proofs are terms

- Function space arrow corresponds to implication; application corresponds to modus ponens

Curry - Howard Isomorphism

- Modus Ponens

$$\frac{A \Rightarrow B \quad A}{B}$$

- Application

$$\frac{\Gamma \mid\!- \; e_1 : \alpha \rightarrow \beta \quad \Gamma \mid\!- \; e_2 : \alpha}{\Gamma \mid\!- \; (e_1 \; e_2) : \beta}$$