Mutually Recursive Types

```ocaml
# 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

```ocaml
# 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

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

A more conventional picture

```
5

7

3  2
```
Mutually Recursive Functions

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

Problem

```ocaml
# 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
```

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

Problem

```ocaml
# 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
```

```ocaml
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 -> 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

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

Nested Recursive Type Values

Ltree = TreeNode(5)
    :|--:|--:|--:[ ]
   |      |      |
TreeNode(3)  TreeNode(2)  TreeNode(5)
    |      |      |
TreeNode(1)  TreeNode(7)
    |      |      |
### Nested Recursive Type Values

![Nested Recursive Type Values](image)

### Mutually Recursive Functions

```ocaml
# 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

```ocaml
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]
```

### 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

- Nested recursive types lead to mutually recursive functions

### Terminology

- **Type**: A type $t$ defines a set of possible data values
  - E.g. `short` in C is $\{x| 2^{15} - 1 \geq x \geq -2^{15}\}$
  - 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, e.g.
  - 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 $\text{op}(\text{arg1}, \ldots, \text{argn})$ 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

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

Type Declarations

- 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: 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 $\Gamma \vdash \text{exp} : t$
- $\Gamma$ is a typing environment
- Supplies the types of variables (and function names when function names are not variables)
- $\Gamma$ is a set of the form $\{ x : \sigma, \ldots \}$
- For any $x$ at most one $\sigma$ such that $(x : \sigma \in \Gamma)$
- exp is a program expression
- $\vdash$ pronounced “turnstyle”, or “entails” (or “satisfies” or, informally, “shows”)

Axioms – Constants (Monomorphic)

- $\Gamma \vdash n : \text{int}$ (assuming $n$ is an integer constant)
- $\Gamma \vdash \text{true} : \text{bool} \quad \Gamma \vdash \text{false} : \text{bool}$
- These rules are true with any typing environment
- $\Gamma, n$ are meta-variables

Axioms – Variables (Monomorphic Rule)

- Notation: Let $\Gamma(x) = \sigma$ if $x : \sigma \in \Gamma$
- **Note**: if such $\sigma$ exists, its unique

Variable axiom:

$$\Gamma \vdash x : \sigma \quad \text{if } \Gamma(x) = \sigma$$

Simple Rules – Arithmetic (Mono)

- Primitive Binary operators ($\oplus \in \{ +, -, *, \ldots \}$):
  $$\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad (\oplus) : \tau_1 \rightarrow \tau_2 \rightarrow \tau_3$$
  $$\Gamma \vdash e_1 \oplus e_2 : \tau_3$$
- Special case: Relations ($\sim \in \{ <, >, =, <=, >= \}$):
  $$\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad (\sim) : \tau \rightarrow \tau \rightarrow \text{bool}$$
  $$\Gamma \vdash e_1 \sim e_2 : \text{bool}$$
- For the moment, think $\tau$ is int

Example: $\{x:\text{int}\} \vdash x + 2 = 3 : \text{bool}$

What do we need to show first?
What do we need for the left side?

$$\{x: \text{int}\} \vdash x + 2 \text{ : int} \quad \{x: \text{int}\} \vdash x + 2 \text{ : bool}$$

Example derivation: if-then-else

$$\Gamma = \{x: \text{int}, \text{int_of_float}: \text{float} \to \text{int}, y: \text{float}\}$$

$$\Gamma \vdash (\text{fun } y \to \begin{array}{l} y > 3 \\ x \quad \Gamma \vdash x + 2 \quad \Gamma \vdash \text{int_of_float } y \text{ : bool} \end{array} : \text{int} \to \text{int})$$

$$\Gamma \vdash (\text{fun } y \to y > 3) \ x$$

then $x + 2$

else $\text{int_of_float } y \text{ : int}$
Function Application

- Application rule:
  \[
  \frac{\Gamma |- e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma |- e_2 : \tau_1}{\Gamma |- (e_1 e_2) : \tau_2}
  \]

If you have a function expression \( e_1 \) of type \( \tau_1 \rightarrow \tau_2 \) applied to an argument \( e_2 \) of type \( \tau_1 \), the resulting expression \( e_1 e_2 \) has type \( \tau_2 \).

Example: Application

\[\Gamma = \{x: \text{int}, \text{int}_\text{of}_\text{float}: \text{float} \rightarrow \text{int}, y: \text{float}\}\]

\[\Gamma |- (\text{fun } y \rightarrow y > 3) : \text{int} \rightarrow \text{bool} \quad \Gamma |- x : \text{int}\]

\[\Gamma |- (\text{fun } y \rightarrow y > 3) x : \text{bool}\]

Fun Rule

- Rules describe types, but also how the environment \( \Gamma \) may change
- Can only do what rule allows!
- fun rule:
  \[
  \frac{\{x: \tau_1\} + \Gamma |- e : \tau_2}{\Gamma |- \text{fun } x \rightarrow e : \tau_1 \rightarrow \tau_2}
  \]

Fun Examples

\[\{y: \text{int}\} + \Gamma |- y + 3 : \text{int}\]

\[\Gamma |- \text{fun } y \rightarrow y + 3 : \text{int} \rightarrow \text{int}\]

\[\{f: \text{int} \rightarrow \text{bool}\} + \Gamma |- f 2 :: [\text{true}] : \text{bool list}\]

\[\Gamma |- (\text{fun } f \rightarrow (f 2) :: [\text{true}]) : (\text{int} \rightarrow \text{bool}) \rightarrow \text{bool list}\]

(Monomorphic) Let and Let Rec

- let rule:
  \[
  \frac{\Gamma |- e_1 : \tau_1 \quad \{x: \tau_1\} + \Gamma |- e_2 : \tau_2}{\Gamma |- (\text{let } x = e_1 \text{ in } e_2) : \tau_2}
  \]

- let rec rule:
  \[
  \frac{\{x: \tau_1\} + \Gamma |- e_1: \tau_1 \quad \{x: \tau_1\} + \Gamma |- e_2: \tau_2}{\Gamma |- (\text{let rec } x = e_1 \text{ in } e_2) : \tau_2}
  \]
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

Modus Ponens

\[
\begin{align*}
A \Rightarrow B & \quad A \\
& \quad B
\end{align*}
\]

Application

\[
\Gamma |- e_1 : \alpha \rightarrow \beta \\
\Gamma |- e_2 : \alpha \\
\Gamma |- (e_1 \cdot e_2) : \beta
\]