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, [])]);
   TreeNode (5, [])]);;

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

Ltree =  TreeNode(5)
          ::                ::                 ::
         [ ]               [ ]
  TreeNode(3)   TreeNode(2)   TreeNode(5)
          ::             ::
         [ ]        [ ]
  TreeNode(1)  TreeNode(7)
          [ ]              [ ]

5
3           2           5
1           7
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;;
```

2/15/24

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

2/15/24

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

2/15/24

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

2/15/24

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

2/15/24
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”

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

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

- **Type judgement** has the form $\Gamma |- \text{exp} : \tau$
- $\Gamma$ 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 $(x: \sigma \in \Gamma)$
- $\text{exp}$ is a program expression
- $\tau$ is a type to be assigned to $\text{exp}$
- $|-\$ pronounced “turnstyle”, or “entails” (or “satisfies” or, informally, “shows”)

Axioms – Constants (Monomorphic)

- $\Gamma |- n : \text{int}$ (assuming $n$ is an integer constant)
- $\Gamma |- \text{true} : \text{bool}$
- $\Gamma |- \text{false} : \text{bool}$
- 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 $\sigma$ exits, its unique

Variable axiom:

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

Simple Rules – Arithmetic (Mono)

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

$$\begin{align*}
\Gamma |- e_1 : \tau_1 & \quad \Gamma |- e_2 : \tau_2 \\
\Gamma |- e_1 @ e_2 : \tau_3
\end{align*}$$

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

$$\begin{align*}
\Gamma |- e_1 : \tau & \quad \Gamma |- e_2 : \tau \\
\quad \tau :) : \tau \to \text{bool} \\
\quad \quad \Gamma |- e_1 \sim e_2 : \text{bool}
\end{align*}$$

For the moment, think $\tau$ is int

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

What do we need for the left side?

$$\begin{align*}
\{x : \text{int}\} |- x + 2 : \text{int} & \quad \{x : \text{int}\} |- 3 : \text{int} \\
\bin & \quad \text{Bin}
\end{align*}$$

What do we need for the left side?

$$\begin{align*}
\{x : \text{int}\} |- x + 2 : \text{int} & \quad \{x : \text{int}\} |- 3 : \text{int} \\
\bin & \quad \text{Bin}
\end{align*}$$
Example: \{x:int\} |- x + 2 = 3 : bool

How to finish?

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

Simple Rules - Booleans

Connectives

\[ \Gamma |- e_1 : \text{bool} \quad \Gamma |- e_2 : \text{bool} \]
\[ \Gamma |- e_1 \&\& e_2 : \text{bool} \]
\[ \Gamma |- e_1 : \text{bool} \quad \Gamma |- e_2 : \text{bool} \]
\[ \Gamma |- e_1 \mid| e_2 : \text{bool} \]

Example derivation: if-then-else-

- \[ \Gamma = \{x:int, \text{int}_\text{of}_\text{float}:\text{float} \to \text{int}, y:\text{float}\} \]
- \[ \Gamma |- (\text{fun} \ y \to \]
  \[ \quad y > 3) \times \quad \Gamma |- x+2 \quad \Gamma |- \text{int}_\text{of}_\text{float} y \]
  \[ \quad : \text{bool} \quad \quad : \text{int} \]
  \[ \quad \Gamma |- \text{if} (\text{fun} \ y \to \ y > 3) \times \]
  \[ \quad \text{then} x + 2 \]
  \[ \quad \text{else} \text{int}_\text{of}_\text{float} y : \text{int} \]

Type Variables in Rules

- If \text{then}_\text{else} \text{rule:}
  \[ \Gamma |- e_1 : \text{bool} \quad \Gamma |- e_2 : \tau \quad \Gamma |- e_3 : \tau \]
  \[ \Gamma |- (\text{if} e_1 \text{then} e_2 \text{else} e_3) : \tau \]
  - \[ \tau \text{ is a type variable (meta-variable)} \]
  - \[ \text{Can take any type at all} \]
  - \[ \text{All instances in a rule application must get same type} \]
  - \[ \text{Then branch, else branch and if}_\text{then}_\text{else} \text{ must all have same type} \]

Function Application

- Application rule:
  \[ \Gamma |- e_1 : \tau_1 \to \tau_2 \quad \Gamma |- e_2 : \tau_1 \]
  \[ \Gamma |- (e_1 \ e_2) : \tau_2 \]
  - \[ \text{If you have a function expression} \ e_1 \text{ of type} \ \tau_1 \to \tau_2 \text{ applied to an argument} \ e_2 \text{ of type} \ \tau_1, \text{ the resulting expression} \ e_1\ e_2 \text{ has type} \ \tau_2 \]
Example: Application

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

\[ \Gamma \vdash (\text{fun } y \rightarrow y > 3) : \text{int} \rightarrow \text{bool} \]

\[ \Gamma \vdash x : \text{int} \]

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

Fun Rule

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

\[
\begin{align*}
\{x \colon \tau_1\} &+ \Gamma \vdash e : \tau_2 \\
\Gamma \vdash \text{fun } x \rightarrow e & : \tau_1 \rightarrow \tau_2
\end{align*}
\]

Fun Examples

\[
\begin{align*}
\{y : \text{int}\} &+ \Gamma \vdash y + 3 : \text{int} \\
\Gamma \vdash \text{fun } y \rightarrow y + 3 : \text{int} \rightarrow \text{int}
\end{align*}
\]

\[
\begin{align*}
\{f : \text{int} \rightarrow \text{bool}\} &+ \Gamma \vdash f \ 2 :: \text{[true]} : \text{bool list} \\
\Gamma \vdash (\text{fun } f \rightarrow (f \ 2) :: \text{[true]}) & : (\text{int} \rightarrow \text{bool}) \rightarrow \text{bool list}
\end{align*}
\]

(Monomorphic) Let and Let Rec

- let rule:

\[
\begin{align*}
\Gamma \vdash e_1 : \tau_1 & \quad \{x : \tau_1\} + \Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash (\text{let } x = e_1 \text{ in } e_2) & : \tau_2
\end{align*}
\]

- let rec rule:

\[
\begin{align*}
\{x : \tau_1\} &+ \Gamma \vdash e_1 : \tau_1 \quad \{x : \tau_1\} + \Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash (\text{let rec } x = e_1 \text{ in } e_2) & : \tau_2
\end{align*}
\]

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**

  
  \[
  A \Rightarrow B \quad A \\
  \quad \quad B
  \]

- **Application**

  
  \[
  \Gamma \vdash e_1 : \alpha \rightarrow \beta \quad \Gamma \vdash e_2 : \alpha \\
  \Gamma \vdash (e_1 \ e_2) : \beta
  \]