CSCI-344
Programming Language Concepts (Section 3)

Lecture 13

Introduction to Type Theory

Instructor: Hossein Hojjat

September 30, 2016

Where we are

Done:

- ML: functional programming language
- Algebraic data types
- Polymorphic Types

This session:

- Introduction to type theory

Types in SML

- SML is **statically** typed
  - Type checking is performed during compile-time (as opposed to run-time)
  - Types are associated with variables not values
- ML is **strongly** typed
  - Every expression has a specific type at compile-time
  - Argument passed to function should closely match the expected type
  - There is no possibility of an unchecked runtime type error
- ML is **implicitly** typed
  - Compiler infers the type of the expression if the program does not specify its type
  - It requires the programmer to specify types only when necessary

Data Types

- **Primitive** data types: a data type provided by the programming language
  - `bool`, `string`, `int`
- **Composite** (compound) data type: a type constructed from other types using type constructors
  - Algebraic type constructors
- **Primitive** type constructor: common classes of algebraic type constructor
  Usually defined in the initial basis of the language
  - Product types (*)
  - Function types (→)
  - List types

Tuples

- SML provides two ways of defining sequences of elements:
  lists and **tuples**
- There are two main differences:
  1. Elements in a list are all of the same type,
     elements of a tuple can be of different types
  2. Number of elements in a tuple type is fixed,
     number of elements in a list can be any nonnegative number
- If the elements of a tuple have types $t_1, t_2, \cdots, t_n$
  then the type of the corresponding tuple is $t_1 * t_2 * \cdots * t_n$

```
- val x = (1,2,3);
> val x = (1, 2, 3): int * int * int
- val y = ("RIT",[2016],1);
> val y = ("RIT", [2016], 1): string * int list * int
```

Tuple Pattern Matching

- Pattern matching decomposes a tuple into its constituent elements

```
val (x,y) = (9,"nine")
val (x, y::ys) = (1,[1,2,3])
```

```
fun coordinate (a,b) = case (a,b) of
  (0, 0) => "origin"
| (0, _) => "on y-axis"
| (_, 0) => "on x-axis"
| _ => "not on any axis";
```

Function Type

- Functions in SML have type ⟨domain type⟩ → ⟨range type⟩
- The most general type of a function: `'a → 'b`
  - Function that takes an argument of some type `'a` and returns a result of some type `'b`

```
- fun cube (x) = x * x * x;
> val cube = fn : int -> int
- fun max (x,y) =  if x < y then y else x;
> val max = fn : int * int -> int
```

SML Types

ty	=	tyvar	type variable
	|	tycon	(nullary) type constructor
	|	ty tycon	(unary) type constructor
	|	(ty, ..., ty) tycon	(n-ary) type constructor
	|	(ty * ... * ty)	tuple type
	|	ty → ty	function type
	|	(ty)	
tyvar	=	'identifier	'a, 'b, ...
tycon	=	identifier	`list, int, bool ...`

Type System

- A type system consists of a set of rules for deriving facts about types of expressions
  - **type equivalence:** when types are the same
  - **type compatibility:** when value of a type can be used
  - **type inference:** what type an expression has

Type Checking

The process of ensuring that a program obeys the type compatibility rules

Type System

Typing judgment: $\Gamma \vdash e : \tau$

- Γ is a typing context: maps names of variables to their types
- τ is a type
- e is an expression

Typing judgment $\Gamma \vdash e : \tau$ is read:

- In type context Γ expression e is well-typed with type τ

Type Checking

Type checking program P is demonstrating the validity of the typing judgment $\vdash P : \tau$ for some type τ

Type System for a Simple Language

```
datatype exp = ARITH of arithop * exp * exp
             | CMP   of relop   * exp * exp
             | LIT   of int
             | IF    of exp     * exp * exp
and      arithop = PLUS | MINUS | TIMES | ...
and      relop   = EQ | NE | LT | LE | GT | GE

datatype ty = INTTY | BOOLTY
```

Type System for a Simple Language

```
datatype exp = ARITH of arithop * exp * exp
             | CMP   of relop   * exp * exp
             | LIT   of int
             | IF    of exp     * exp * exp
and      arithop = PLUS | MINUS | TIMES | ...
and      relop   = EQ | NE | LT | LE | GT | GE

datatype ty = INTTY | BOOLTY
```

Examples to rule out:

- Can't add an integer and a boolean
  - `- 1 + (2 > 3)`
  - `- (ARITH(PLUS, LIT 1, CMP (GT, LIT 2, LIT 3)))`
- Can't compare an integer and a boolean
  - `- (1 < (2 = 3))`
  - CMP (LT, LIT 1, CMP(EQ (LIT 2, LIT 3)))

Rule for Arithmetic Operators

Informal example:

$$\frac{\vdash 1 : \texttt{int} \qquad \vdash 2 : \texttt{int}}{\vdash 1 + 2 : \texttt{int}}$$

General form:

$$\frac{\vdash e_1 : \texttt{int} \qquad \vdash e_2 : \texttt{int}}{\vdash \texttt{ARITH}(\_, e_1, e_2) : \texttt{int}}$$

Rule for Comparisons

Informal example:

$$\frac{\vdash 5 : \texttt{int} \qquad \vdash 4 : \texttt{int}}{\vdash 5 < 4 : \texttt{bool}}$$

General form:

$$\frac{\vdash e_1 : \texttt{int} \qquad \vdash e_2 : \texttt{int}}{\vdash \texttt{CMP}(\_, e_1, e_2) : \texttt{bool}}$$

Rule for Literals

Informal example:

$$\vdash 5 : \texttt{int}$$

General form:

$$\vdash \texttt{LIT}(n) : \texttt{int}$$

Rule for Conditionals

Informal example:

$$\frac{\vdash \texttt{true} : \texttt{bool} \qquad \vdash 6 : \texttt{int} \qquad \vdash 42 : \texttt{int}}{\vdash \texttt{IF}(\texttt{true}, 6, 42) : \texttt{int}}$$

General form:

$$\frac{\vdash e : \texttt{bool} \qquad \vdash e_1 : \tau_1 \qquad \vdash e_2 : \tau_2 \qquad \tau_1 = \tau_2}{\vdash \texttt{IF}(e, e_1, e_2) : \tau_1}$$

Type Checker in ML

```
exception IllTyped;
fun tc (ARITH (_, e1, e2)) =  case (tc e1, tc e2)  of
        (INTTY, INTTY) => INTTY
        | _                => raise IllTyped
   | tc (CMP (_, e1, e2)) => case (tc e1, tc e2) of
        (INTTY, INTTY) => BOOLTY
        | _                 => raise IllTyped
   | tc (LIT _) => INTTY
   | tc (IF (e,e1,e2)) => case (tc e, tc e1, tc e2) of
        (BOOLTY, tau1, tau2) =>
           if eqType(tau1, tau2)
           then tau1 else raise IllTyped
   | _                      => raise IllTyped;
```

Summary

This Lecture

- Introduction to type theory

Next Lecture

- Type Systems, Typed Impcore