Unit-V Application of Functional Programming in λ Calculus

September 21, 2015

Functional Programming in Standard ML

- Introduction
  - ML(Meta Language) is a general purpose language with a powerful functional subset
  - It is used mainly as a design and implementation tool for computing theory based research and development
  - It is also used as a teaching language
  - ML is strongly typed with compile time type checking
  - Function calls are evaluated in applicative order

Functional Programming in Standard ML

- ML originated in the mid 1970s as a language for building proofs in Robin Milners LCF(Logic for Computable Functions) computer assisted formal reasoning system
- SML(Standard ML) was developed in the early 1980s from ML with extensions from the Hope functional language
- SML is one of the first programming languages to be based on well defined theoretical foundations
- SML system usage and show the result of evaluating an expression
  - -$<$ *expression* $>$;
  - $>$ $<$ *result* $>$

Types

- Types are central to SML
- Every object and construct is typed
- Unlike Pascal, types need not be made explicit but they must be capable of being deduced statically from a program
- SML provides several standard types, for example for booleans, integers, strings, lists and tuples
- SML also has a variety of mechanisms for defining new types but not considered in this course(May be required for Practical's – Please refer internet or ebook)

Types

- When representing objects, SML always displays types along with values:
- $<$ *value* $>$: $<$ *type* $>$
- In particular, function values are displayed as: fn : $<$ *type* $>$

Types

- ML has a rich collection of data types. We can divide the collection of data types into three categories
  - Basic data types: ML has six basic data types: integer, string, character, boolean, real, and unit
  - Structured data types: Type operators combine types to form structured, or compound, types. Three built-in type operators:
    1. tuples, records, and lists
    2. Another built-in type operator for functions
  - User-defined types: The user-defined data types are variant record types found in other programming languages. Variant records are not used much in other programming languages, but user-defined types are quite important to programming in ML.

Lists

- A list must contain elements of the same type and end with the empty list
- Lists cannot be used to represent records with different type field
- Lists are written as , separated element sequences within [and]
- There is an implied empty list at the end of a list
- The type expression for a list depends on the element type: $< elementtype >$ list

Lists Contd..

- The notation with the square brackets is just a special syntax of building up lists using the constructors :: and nil
- It is important in SML to distinguish constructors from ordinary functions
- Constructors are the primitive functions that create new values of a data
- For example, the constructors for lists are :: and nil
- The append function @ is not a constructor (it can be defined using :: and nil

Examples

1. Input:- [1,4,9,16,25]; Output: val it = [1,4,9,16,25] : int list
2. Input:- ["ant","beetle","caterpillar","dragonfly","earwig"]; val it = ["ant","beetle","caterpillar","dragonfly","earwig"] : string list
3. Input:[[1,1],[2,8],[3,27],[4,64],[5,125]]; Output: [[1,1],[2,8],[3,27],[4,64],[5,125]] : (int list) list

- The use of (and) to structure the type expression

Tuples

- An ML tuple, like a Pascal RECORD, is fixed length sequence of elements of different types, unlike a list which is variable length sequence of elements of the same type
- Tuples are written as , separated sequences within (and)
- Tuples are structured data types of heterogeneous elements listed in order

Examples

1. Input:("VDUs",250,120); Output:val it = ("VDUs",250,120) : string * int * int
2. Input:[(("A","B"),"Accounts",101),(("C","D"),"Office",102)]; Output:val it = [(("A","B"),"Accounts",101),(("C","D"),"Office",102)] : ((string * string) * string * int) list

Function Types and Expressions

- A function uses values in an argument domain to produce a final value in a result range
- In SML, a functions type is characterised by its domain and range types: fn : $< domaintype > \rightarrow < rangetype >$
- Tuples are normally used to enable uncurried functions with multiple bound variables
- In SML, as in λ calculus and LISP, expressions are usually based on prefix notation function applications with the function preceding the arguments:$< functionexpression >< argumentexpression >$
- Function applications are evaluated in applicative order

Function type and expressions

- Function applications need not be explicitly bracketed but brackets should be used round arguments to avoid ambiguity
- SML enables uncurried binary functions to be used as infix operators so the function name may appear in between the two arguments
- They are then typed as if they had tuples for arguments
- Many standard binary functions are provided as infix operators. They may be treated as prefix functions on tuples by preceding them with: op

Boolean Standard functions

- Many standard binary functions are provided as infix operators. They may be treated as prefix functions on tuples by preceding them with:
- The boolean negation function: not
- Returns the negation of its boolean argument
- Conjunction and disjunction are provided through the sequential infix operators:andalso orelse
- SML systems may not be able to display these operators types but they are effectively: fn : (bool * bool) $\rightarrow$ bool as they both take two boolean arguments, which are treated as a: bool * bool tuple for infix syntax, and return a boolean result

Numeric standard functions and operator overloading

- SML provides real numbers as well as integers
- Same operators are used for both even though they are distinct types
- The use of the same operator with different types is known as operator overloading
- The addition, subtraction and multiplication infix operators are:+,-,*
- SML systems may not display their types because they are overloaded. SML literature uses the invented type: num
- The above indicate both integer and real so these operators types might be: fn : (num * num) $\rightarrow$ num as they take two numeric arguments, with infix syntax for a tuple, and return a numeric result

Numeric standard functions and operator overloading

- Note: For each operator both arguments must be the same type
- Integer Division:
  - div is for integer division
  - We can use op to convert it to prefix form to display its type:
  - Command: op div; Result: val it = fn : int * int $\rightarrow$ int
  - Example: Input:6 * 7 div (7 - 4) + 28; Result:val it = 42 : int
  - Negation Operator is $\sim$

String Standard Functions

- Binary Infix Operator: ^: Concatenates two strings together
- Command: op^; Output: val it = fn : string * string $\rightarrow$ string
- Example: Input:"Happy" ^"birthday!"; Output:val it = "Happybirthday!" : string
- size: returns the size of a string
- Command: size; Output:val it = fn : string $\rightarrow$ int
- Example: size "SASTRA"; val it = 6 : int

List Standard Functions

- In SML, lists are accessed by the head and tail operators:hd tl
- Example: Command: hd; Output: fn : (a list) -¿ a
- Example: Command: hd [1,2,3,4,5]; Output: 1 : int
- Example: Command: tl; Output: fn : (a list) -¿ (a list)
- Example: Command: tl ["alpha","beta","gamma","delta","epsilon"]; Output: ["beta","gamma","delta","epsilon"] : string list
- The infix list concatenation operator is: ::
- Given an object and a list of the same type of object, :: returns a new list with the object in the head and the object list in the tail
- Example: Command: 0::[1,2,3,4,5]; Output: [0,1,2,3,4,5] : int list

List Standard Functions

- The operators hd, tl and :: are said to be polymorphic because they apply to a list of any type of object

Characters, Strings and Lists

- SML does not provide a separate character type. Instead, a character is a one letter string
- The standard function ord converts a single character string to the equivalent ASCII code
- Command: ord; fn : string $\rightarrow$ int
- Example: Command: ord "a"; Output: 97 : int
- chr: converts an integer ASCII value into the equivalent single character string. Command: chr; fn : int $\rightarrow$ string
- Example: Command: chr 54; Output: "6" : string

Characters, Strings and Lists

- explode: Access the individual characters making up a string it must be unpacked into a list of single character. explode : string $\rightarrow$ (string list)
- Example: Command: explode "hello"; Output: ["h","e","l","l",o"] : string list strings
- implode: converts a list of strings to a single string. implode:(string list) $\rightarrow$ string

Comparison Operators

- $<, >, =, \geq, \leq, <>$
- SML systems may not display these operators types because they are overloaded
- For all these operators, both arguments must be of the same type
- Example: Command: "SASTRA"<"sastra"; Output: val it = true : bool

Functions

- fn $< boundvariables > \implies < expression >$
- A bound variable is known as an alphabetic identifier and consists of one or more letters, digits and _s starting with a letter
- Examples:
  - Command: fn x $\implies$ x+1; Output: fn : int $\rightarrow$ int
  - Command:fn x $\implies$ fn y $\implies$ not (x orelse y); Output: fn : bool $\rightarrow$ (bool $\rightarrow$ bool)

Making bound variables types explicit

- Suppose we try to define a squaring function:fn x $\implies$ x*x
- Because * is overloaded, SML cannot deduce xs type and will reject this function
- Domain types may be made explicit by following each bound variable with its type. Thus for a single bound variable: (<bound variable> : <type>)
- Example: Command:fn (x:int) $\implies$ x*x; Output:val it = fn : int $\rightarrow$ int
- For a tuple of bound variables:(<bound variable1> : <type1>, <bound variable2> : <type2>, ...)
- Example: Command: fn (x:int,y:int) $\implies$ x*x+y*y;Output:val it = fn : int * int $\rightarrow$int

Global Definitions

- Global definitions may be established with:val <name> = <expression>
- Example:Command:val sq = fn (x:int) $\Longrightarrow$ x*x; Output:val sq = fn : int $\rightarrow$ int
- Defined names may be used in subsequent expressions
- Example: Command:sq 3; Output:val it = 9 : int
- Example:val sum_sq = fn (x:int,y:int) $\Longrightarrow$ (sq x)+(sq y)

Conditional Expressions

- The SML conditional expression has the form:if <expression1> then <expression2> else <expression3>
- The first expression must return a boolean and the option expressions <expression2> and <expression3> must have the same type
- Example:Command:val max = fn (x:int,y:int) $\Longrightarrow$ if x>y then x else y; Output:val max = fn : (int * int) $\rightarrow$ int
- Example:Command:val imp = fn (x,y) $\Longrightarrow$ if x then y else true; Output:val imp = fn : bool * bool $\rightarrow$ bool

Recursion and Functions