Unit-V Application of Functional Programming in λ Calculus

October 16, 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: ˆ: Concatenates two strings together
- Command: opˆ; Output: val it = fn : string * string → string
- Example: Input:"Happy" ˆ"birthday!"; Output:val it = "Happybirthday!" : string
- size: returns the size of a string
- Command: size; Output:val it = fn : string → int
- Example: size "SASTRA"; val it = 6 : int

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

- fn x $\Longrightarrow$ x*x
- * is overloaded, SML cannot x's type and will reject function
- Explicit domain types $< boundvariable >:< type >$
- Example: fn(x:int) $\Longrightarrow$ x*x; Output: fn: int $\rightarrow$ int
- For tuples, $(< var1 >:< type1 >, < var2 >:< type2 >, ...)$
- fn(x:int,y:int) $\Longrightarrow$ x*x+y*y; fn:int*int$\rightarrow$int

Global Definition

- val $< name >=< expression >$
- Ex: val sq=fn(x:int) $\Longrightarrow$ x*x;
- Defined names can be used in subsequent expressions
- val sum_sq=fn(x:int,y:int) $\Longrightarrow$ (sq x)+(sq y)

Conditional Expressions

- if $< exp1 >$ then $< exp2 >$ else $< exp3 >$
- exp1 evaluates to bool. exp2 and exp3 must be of the same type
- Ex: Maximum of 3 numbers
- Ex: implication

Recursion and Function Definition

- To define recursion use keyword rec
- Length of list
- Shorthand for of function definition
  - Instead of val use fun
  - fn, $\Longrightarrow$ dropped
  - Bound variables are moved to the left
  - fun $<$ *name* $><$ *boundvariable* $>=<$ *exp* $>$
  - val rec $<$ *name* $>=$fn$<$ *boundvariable* $>$ $\Longrightarrow$ $<$ *expression* $>$
- Example: Finding squares of a list of numbers
- Example: Inserting a string in the beginning of a list
- Example: gcd,abs,exp and mutual recursion

Tuple Selection

- Tuple elements are selected by defining functions with appropriate bound variable tuples
- Example: Command: fun tname (n:(string * string),d:string,p:int) = n; Output: val tname = fn : (string * string) * string * int → string * string
- To avoid writing out bound variables which are not used in the function body, SML provides the wild card variable: _ which behaves like a nameless variable of arbitrary type
- Example:Command:fun tname (n:(string * string),_, _) = n; Output: val tname = fn : (string * string) * 'a * 'b → string * string
- SML uses 'a and 'b to stand for possibly distinct unknown types

Pattern Matching

- It is common SML practise to use case style function definitions with pattern matching rather than conditional expressions in a function's body
- These are known as clausal form definitions
- The general form is:fun $< name >< pattern1 >=< expression1 > \mid < name >< pattern2 >=< expression2 > \mid ... < name >< patternN >=< expressionN >$
- Here, each: $< name >< patternI >=< expressionI >$ defines a case
- Note that the order of the cases is significant

Pattern Matching

- When a case defined function is applied to an argument, each pattern is matched against the argument in turn, from first to last, until one succeeds
- The value of the corresponding expression is then returned
- Example: schools.sml
- Example: listleng.sml
- Example: cubelist.sml
- Example: stri.sml
- The above function is a curried function

Local Definitions

- SML uses the let ... in ... notation for local definitions: let val $< name >=< expression1 >$ *in* $< expression2 >$ end
- This evaluates $< expression2 >$ with $< name >$ associated with $< expression1 >$
- For function definitions: let fun $< name >< pattern >=< expression1 >$ *in* $< expression2 >$ end and the corresponding case form is used

New Types

- A new concrete type may be introduced by a datatype binding
- This is used to define a new type's constituent values recursively by
  1. Listing base values explicitly
  2. Defining structured values in terms of base values and other structured values
- The binding introduces new type constructors which are used to build new values of that datatype
- They are also used to identify and manipulate such values

New Types Contd..

- datatype $< constructor >=< constructor1 > | < constructor2 > |... < constructorN >$
- which defines the base values of type ¡constructor¿, an identifier, to be the type constructor identifiers $< constructor1 > or < constructor2 >$ etc