CSCI: 4500/6500 Programming Languages

Functional Programming Languages

Part 4: Standard Meta Language (SML) & Haskell

Standard ML

- ML - historically stands for Meta Language. ML was a meta language for expressing and manipulating logical proofs.
- General purpose, modular functional programming language developed a team in the 1970s at the University of Edinburgh, headed by Robin Milner (polymorphism paper in reading list).
  - » Polymorphism – one behavior for different types
  - » First language to include "polymorphic type inference" (functions with multiple different types – different input parameters) together with a type-safe exception handling mechanism.

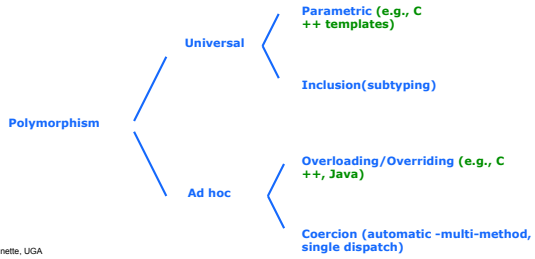
Preview: Polymorphism

- One behavior (e.g., a function definition) for multiple different types (e.g., a function handles different types of input parameters).

- Ad-Hoc Polymorphism: range of actual types is finite and the combinations must be specified individually prior to use so multiple definitions (overloading, coercision) -> compiler calls the right definition
- Parametric Polymorphism (first type of polymorphism to appear in an actual programming language – ML in 1976)
  - » NO explicit type definition (e.g., the append function of a list)
  - » Used transparently with any number of types
  - » Generic Programming (arguably): Only one definition (e.g., templates, macro, note instantiation is laziness, "not evaluated until needed" characteristics)
- Suptyping Polymorphism (inheritance) – classes related by supertype.

Polymorphism

- Universal polymorphism : Allows writing code that works with different types
- Ad-hoc polymorphism : Selecting the right implementation code to be executed



Standard ML

- Uses type declarations, but also does type inferencing to determine the types of undeclared variables
  - » type of all variables can be determined at compile time.
  - » function Foo(a b) = a + b
- Static-scoped
- Syntax is closer to Pascal than to LISP
  - » e.g., infix arithmetic expressions instead of Cambridge postfix
- Restrictions on how data types are intermixed (more later):
  - » Example: integer division may not be used on strings
  - » ML is strongly typed (whereas Scheme is essentially typeless) and has no type coercions (talk more about this later in the Semester)
- Includes exception handling
- Module facility for implementing abstract data types.
- Permits side-effect (therefore an *impure* functional language)

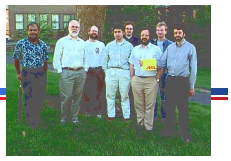
Standard ML (cont)

- Standard ML is a domain-specific language that is appropriate for *building* compilers
- Support for
  - » Complex data structures (abstract syntax, compiler intermediate forms)
  - » Memory management like Java
  - » Large projects with many modules
  - » Advanced type system for error detection

Learn more details

- Today we will cover the basics so you can get started.
- Resources:
  - » Robert Harper's (pdf book)
    - – http://www.cs.cmu.edu/~rwh/smlbook/online.pdf
  - » Peter Lee's:
    - – http://www.cs.cmu.edu/afs/cs/local/sml/common/smlguide/smlnj.htm
  - » SML/NJ Literature:
    - – http://www.smlnj.org/doc/literature.html#tutorials
    - – Runs on Microsoft Windows, MacOS X (yay!), UNIX,
  - » Short & concise tutorial (not available)
    - – http://cs.wwc.edu/Environment/SML-Tutorial.html

Installation



Distribution (SML of New Jersey):

- http://www.smlnj.org/dist/working/110.69/index.html
- Developed at Bell laboratories and Princeton University
- Installation (straight forward): Set your `PATH` variable where you install it.
  - » Forgot?
    - – `find / -name sml –print`
    - – `/usr/local/smlnj-110.69/bin/sml # default on a MAC`
    - – `export PATH=$PATH:/usr/local/smlnj-110.69/bin`
    - – `setenv PATH $PATH:/usr/local/smlnj-110.69/bin`
- Run:

```
{saffron:ingrid:219} sml
Standard ML of New Jersey v110.69 [built: Tue Feb
  3 22:24:07 2009]
-
```


ML

Interactive

- Type in expressions
- Evaluate and print type and result
- End with ;
- Exit (enter end of file)

Compileable

Hello word! in SML

```
- print("Hello world!\n");
Hello world
val it = () : unit
-
```

"it" is the default name of the expression.

Preliminaries

- Read – Eval – Print – Loop:

```
- 3+2;
```


Preliminaries

- Read – Eval – Print – Loop:

```
- 3+2;
val it = 5 : int
```

Preliminaries

- **Read – Eval – Print – Loop:**

```
- 3+2;
val it = 5 : int
- it + 7 ;
```


Preliminaries

- **Read – Eval – Print – Loop:**

```
- 3+2;
val it = 5 : int
- it + 7 ;
val it = 12 : int
```


Preliminaries

- **Read – Eval – Print – Loop:**

```
- 3+2;
val it = 5 : int
- it + 7 ;
val it = 12 : int
- it - 3 ;
```


Preliminaries

- **Read – Eval – Print – Loop:**

```
- 3+2;
val it = 5 : int
- it + 7 ;
val it = 12 : int
- it - 3 ;
val it = 9 : int
```


Preliminaries

- **Read – Eval – Print – Loop:**

```
- 3+2;
val it = 5 : int
- it + 7 ;
val it = 12 : int
- it - 3 ;
val it = 9 : int
- 4 + true
```


Preliminaries

- **Read – Eval – Print – Loop:**

```
- 3+2;
val it = 5 : int
- it + 7 ;
val it = 12 : int
- it - 3 ;
val it = 9 : int
- 4 + true
= ;
stdIn:14.1-14.9 Error: operator and operand don't
   agree [literal]
  operator domain: int * int
  operand:         int * bool
  in expression:
    4 + true
```

- Copy and paste the following text into a Standard ML window:

```
2+2;          (* note semicolon at end*)
3*4;
4/3;          (* an error! *)
6 div 2;      (* integer division *)
7 div 3;
```



```
- 4/3 ;
stdIn:20.1-20.4 Error: operator and operand
  don't agree [literal]
  operator domain: real * real
  operand:         int * int
  in expression:
    4 / 3
- 4.0 / 3.0 ;
val it = 1.33333333333 : real
```


List functions

- [1,2,3,4];

-val it : [1,2,3,4] : int list

- val myList = [1,2,3,4];

-val myList : [1,2,3,4] : int list

- 0 :: [1,2, 3];

-val it = [0,1,2,3] : int list - 213 :: 0 :: [1,2, 3];val it = [213,0,1,2,3] : int list

- Includes lists and list operations
- The `val` statement binds a name to a value (similar to `DEFINE` in Scheme)
- Function declaration form:

  `fun` function_name (formal_parameters) =

  function_body_expression;

e.g.,

```
fun cube(x : int) = x * x * x ;
fun square( x: int) : int = x * x ;
```


Haskell

- Similar to ML (syntax, static scoped, strongly typed, type inferencing)
- Different from ML (and most other functional languages) in that it is purely functional (e.g., no variables, no assignment statements, and no side effects of any kind)

Haskell

- Most Important features
  - » Uses lazy evaluation (evaluate no subexpression until the value is needed)
  - » Has list comprehensions, which allow it to deal with infinite lists

Our First Program

- ghci
- “Hello World “
- putStrLn "Hello World”

Compile:

ghc -o hello hello.hs

./hello

Function Definition

» let fac n = if n == 0 then 1 else n * fac (n-1)

- fac 10

Haskell

- Next project: you can choose Haskell or SML.
- For Haskell you are expected to use the Glasgow compiler (co-dependencies – perl, gcc) -> compiles like C (ghc –o main main.hs)
- Other compilers: Glasgow (Glorious), Helium, Hugs, Omega (is strict).

Concurrent Haskell

- ghc spare --make -threaded
- Enable threads:
  - » time ./primes-test +RTS -N2
- Run Example Program in threaded and unthreaded mode
  - » “True Parellelism”
  - » Running threads in parallel & multiple processors
    - – Pitfalls?

Applications of Functional Languages

- APL is used for throw-away programs
- LISP is used for artificial intelligence
  - » Knowledge representation
  - » Machine learning
  - » Natural language processing
  - » Modeling of speech and vision
- Scheme is used to teach introductory programming at a significant number of universities

Comparing Functional and Imperative Languages

- Imperative Languages:
  - » Efficient execution
  - » Complex semantics
  - » Complex syntax
  - » Concurrency is programmer designed
- Functional Languages:
  - » Simple semantics
  - » Simple syntax
  - » Inefficient execution
  - » Programs can automatically be made concurrent

Functional Programming in Perspective (pros)

- Advantages of functional languages
  - » lack of side effects makes programs easier to understand
  - » lack of explicit evaluation order (in some languages) offers possibility of parallel evaluation (e.g. MultiLisp)
  - » lack of side effects and explicit evaluation order simplifies some things for a compiler (provided you don't blow it in other ways)
  - » programs are often surprisingly short
  - » language can be extremely small and yet powerful

Functional Programming in Perspective (cons)

- Advantages of functional languages
  - » difficult (but not impossible!) to implement efficiently on von Neumann machines
    - – lots of copying of data through parameters
    - – (apparent) need to create a whole new array in order to change one element
    - – heavy use of pointers (space/time and locality problem)
    - – frequent procedure calls
    - – heavy space use for recursion
    - – requires garbage collection
    - – requires a different mode of thinking by the programmer
    - – difficult to integrate I/O into purely functional model