CSC 372, Spring 2025

Intro to Standard ML continued and Types Intro

Michelle Strout

THE UNIVERSITY OF ARIZONA

January 23, 2025

Plan

- **Announcements**
  - SA2 is due Wednesday Jan 29th, you will be writing 10 SML functions
- **Last time**
  - Review some pre-assessment quiz questions
  - ICA2: Quiz on the syllabus
  - Functional programming and SML intro
- **Today**
  - Questions from you all
  - ICA3: Participation quiz on pattern matching and types
  - SML intro continued and Types intro

TopHat Questions

- **ICA2: Syllabus**
  - Go to gradescope to see how you did, see piazza announcement
  - The class median on the 10 questions was 8/10
- **Some pre-assessment questions**
  - Python's type system, 28/54 people got this correct
  - Java's type system, 25/54 people got this correct
- **Link languages to motivation in TopHat**
  - Kotlin, Java
  - C, C#
  - JavaScript, Scala
  - Prolog, GoLang
  - Chapel, Lisp

Pace of the course

- **DRC**
  - Accessibility and Accommodations: At the University of Arizona, we strive to make learning experiences as accessible as possible. If you anticipate or experience barriers based on disability or pregnancy, please contact the Disability Resource Center (520-621-3268, https://drc.arizona.edu) to establish reasonable accommodations.
  - Dr. Strout is happy to work with the DRC and any of you to help you achieve the learning objectives for this course.
- **TopHat Question about class pace**
- **Suggestions**
  - Ask us questions after class, in piazza during class or whenever, in office hours, it helps the rest of the class as well as you!
  - Ask AIs questions any time (except when taking a quiz or exam)
  - Evaluate all answers you receive from staff, AI, and the internet

ICA3: Participation quiz on SML and Types

THE UNIVERSITY OF ARIZONA
Computer Science

- **Read the instructions on the quiz**

Functional Programming and SML Outline

- Functions and pattern matching in SML (Hands On)
- Recursion in SML (Hands On)
- Unit testing and exceptions in SML (Hands On)
- Other things to try in SML
- Intro to Types: motivation and terminology

Pattern matching: Writing/executing SML code

- **Steps**
  1. Go to Piazza and then syllabus on GitHub
  2. Git clone the course materials repository which has Sandboxes/
  3. Assume Docker desktop has already been installed (SA1)
  4. 'cd' into the repository in a terminal and in vscode terminal
  5. Start the docker container in the terminal
  6. Edit files in vscode (or favorite editor)

- **Code**

```
/workspace$ poly                  // start sml interpreter
Poly/ML 5.7.1 Release
> fun foo 0 = 0
#   | foo _ = ~1;
> foo 3;
```

Intro to Pattern Matching

- **What is Pattern Matching?**
  - A mechanism to destructure and analyze data.
- **Why Pattern Matching**
  - Simplifies, and often eliminates, complex conditionals.
  - Makes code more readable and declarative.
- **Functional programming context**
  - Common in languages like SML, Haskell, and OCaml

Collab Reference: This slide and next 5 slides generated with help of ChatGPT on Jan 23, 2025

Basic Syntax in SML

- **Pattern Matching in 'case' Expressions**

```
case expr of
     pattern1 => result1
   | pattern2 => result2
   | ...
```

- **Pattern Matching in Function Definitions**

```
fun f pattern1 = result1
  | f pattern2 = result2
```

- **Example to try in SML interpreter**

```
> fun factorial 0 = 1
#   | factorial n = n * factorial (n - 1);
> factorial 3;
```

Function Clause Matching and Var Bindings

- **How functions match patterns**
  - Expression passed into a function is matched against the patterns in order.
  - The first matching pattern determines which clause is executed.

- **Variable Binding**
  - When a pattern matches, variables in the pattern are bound to corresponding parts of the expression.

```
fun greet (name, age) = "Hello, " ^ name ^ ", age " ^ Int.toString age
```

  - Example call: `greet (“Alice”, 30)`
    - Binds `name = “Alice”`, `age = 30`

- **Implications**
  - Encourages concise and expressive code.
  - Avoids explicit conditional logic.

Patterns in Depth

- **Wilecard: Matches any value, but ignores it**

```
fun f _ = "ignored"
```

- **Tuples**

```
fun add (x, y) = x + y
```

- **Lists**

```
fun sum [] = 0
  | sum (x::xs) = x + sum xs
```

- **Records**

```
fun greet {name, age} = "Hello " ^ name
```

Benefits and Common Use Cases

- **Error prevention**
  - Ensures all cases are handled (compiler warnings for missing ones)
- **Declarative style**
  - Code expresses what to do rather than how.
- **Examples in practice**
  - Parsing structured data.
  - Implementing interpreters.
  - Simplifying algorithms with recursive data types.

Advanced Topics and Limitations

- **Nested patterns**

```
fun deepMatch ((x, y), [z]) = x + y + z
```

- **Guards**

```
fun describe x =
      case x of
          n when n > 0 => "Positive"
        | n when n < 0 => "Negative"
        | _ => "Zero"
```

- **Limitations**
  - Non-exhaustive patterns.
  - Complexity for large patterns.

Pattern Matching in SML

- **Key Concepts**
  - A concise and expressive way to destructure and analyze data.
  - Patterns can include wildcards '_', literal values, and nested patterns
  - Pattern matching is exhaustive, all cases must be covered

- **Code**

```
(* Pattern matching for natural numbers *)
fun factorial 0 = 1
  | factorial n = n * factorial (n - 1);

(* Pattern matching with list recursion *)
fun sumList [] = 0
  | sumList (x :: xs) = x + sumList xs;
```

- **Questions**
  - What is 'xs' being bound to?

Recursion instead of iteration

- **No loops in SML, Recursion is used instead**
  - At least one base case will be needed
  - The recursive part of the function needs to make progress to avoid an infinite "loop"

- **Code: imitating a loop with recursion**

```
fun loop i n =
    if i > n then ()    (* Base case *)
    else (
        print (Int.toString i ^ "\n");
        loop (i + 1) n (* Recurse: increment i *)
    );
```

- **Try this in interpreter**

What is wrong with recursion in these?


```
fun removeNegatives (x :: xs) =
      let
          val result = removeNegatives xs
      in
          if x < 0 then result
          else x :: result
      end;
```

```
fun decrList [] = []
  | decrList (x :: xs) = (x-1) :: decrList xs;
```

Testing your SML functions

- **See sml-intro-in-class.sml**
  - Uncomment the 'use "Unit.sml"; ' code
  - Can check expected results and if an exception has occurred

- **Code: Some example usage**

```
val () =
  Unit.checkExnWith Int.toString
  "minlist [] should raise an exception"
  (fn () => minlist [])

val () =
  Unit.checkExpectWith
  (Unit.listString (Unit.pairString Int.toString Int.toString))
  "zip ([],[]) should be []"
  (fn () => zip ([],[]))
  []
```

Exceptions Example in SML

- **Code**

```
fun drop 0 l = l
  | drop n [] = raise ListTooShort
  | drop n (x::xs) = drop (n-1) xs

val res7 = drop 2 [1,2,3,4]
val res8 = drop 3 [10, 20, 30]
           handle ListTooShort =>
              (print "List too short!"; [])
```

- **Questions**
  - What is res7 going to be?
  - What is res8 going to be?

Other things to try

- **In the poly REPL, try the following:**

```
let x=3 and y=4 in x+y end;  (* poly REPL balks *)
real;
explode;
ord;
trunc;
floor;
ceil;
round
chr;
str;
(op +);
```

- **Questions for you to answer**
  - What do each of the above do?
  - Ask an AI how to fix the error you get for the first one.

Why Types Matter

- **Motivation for Type Checking**
  - Types catch errors at compile-time, reducing runtime bugs.
  - Improves program readability and maintainability.
- **Examples of mistakes Types can prevent**
  - Passing a string where a number is expected.
  - Misuing functions or operators.
- **Functional programming context**
  - Common in languages like SML, Haskell, and OCaml

Collab Reference: This slide and next 4 slides generated with help of ChatGPT on Jan 23, 2025

Key Terminology

- **Strong vs. Weak Typing**
  - Strong: Enforces strict type rules (e.g., SML, Haskell).
  - Weak: Allows implicit type conversions (e.g., JavaScript).
- **Static vs. Dynamic Typing**
  - Static: Types are checked at compile-time (e.g., SML).
  - Dynamic: Types are checked at runtime (e.g., Python).
- **Inferred vs. Annotated Types**
  - Inferred: Types deduced by the compiler or runtime (e.g., SML, Python).
  - Annotated: Types explicitly specified by the programmer (e.g., Java, C, C++, Chapel)

Void and Unit Types

- **Void Types**
  - Represents "no value" or "nothing".
  - Often used in imperative languages (e.g., `void` in C).
- **Unit Type in Functional Languages**
  - Represents a single value: '( )'.
  - Commonly used where a value is required but irrelevant.

```
fun printHello () = print "Hello"
```

What's Coming in Future Classes

- **Polymorphism**
  - Parametric polymorphism (e.g., generics).
  - Ad-hoc polymorphism (e.g., function/method overloading).
- **Classes and Types**
  - Object-oriented concepts like classes and interfaces.
  - Runtime polymorphism through mechanisms like method overriding and dynamic dispatch
- **Advanced Topics**
  - Dependent typing, gradual typing, and typeclasses.
  - Design choices in string and array types.
  - Mixins, protocols, and type erasure.

Summary and Takeaways on Type Intro

- **Types protect programmers**
  - Compile-time checks catch many errors early.
- **Understanding key concepts**
  - Strong vs. weak
  - Static vs. dynamic
  - Inferred vs. annotated
- **Preview of advanced topics**
  - Future classes will cover polymorphism, type inference, some type theory, and type choices.