Introduction to SML

Getting Started

Michael R. Hansen

mrh@imm.dtu.dk

Informatics and Mathematical Modelling

Technical University of Denmark

Some Background on Functional Programmin

In functional programming, the model of computation is the application of functions to arguments. no side-effects

Some Background on Functional Programmin

In functional programming, the model of computation is the application of functions to arguments. no side-effects

- Introduction of λ-calculus around 1930 by Church and Kleene when investigating function definition, function application, recursion and computable functions. For example, $f(x) = x + 2$ is represented by $\lambda x.x + 2$.

Some Background on Functional Programmin

In functional programming, the model of computation is the application of functions to arguments.

no side-effects

- Introduction of λ-calculus around 1930 by Church and Kleene when investigating function definition, function application, recursion and computable functions. For example, $f(x) = x + 2$ is represented by $\lambda x.x + 2$.
- Introduction of the type-less functional-like programming language LISP was developed by McCarthy in the late 1950s.

Some Background on Functional Programmin

In functional programming, the model of computation is the application of functions to arguments. no side-effects

- Introduction of λ-calculus around 1930 by Church and Kleene when investigating function definition, function application, recursion and computable functions. For example, $f(x) = x + 2$ is represented by $\lambda x.x + 2$.
- Introduction of the type-less functional-like programming language LISP was developed by McCarthy in the late 1950s.
- Introduction of the "variable-free" programming language FP (Backus 1977), by providing a rich collection of functionals (combining forms for functions).

Some Background on Functional Programmin

In functional programming, the model of computation is the application of functions to arguments.

no side-effects

- Introduction of λ-calculus around 1930 by Church and Kleene when investigating function definition, function application, recursion and computable functions. For example, $f(x) = x + 2$ is represented by $\lambda x.x + 2$.
- Introduction of the type-less functional-like programming language LISP was developed by McCarthy in the late 1950s.
- Introduction of the "variable-free" programming language FP (Backus 1977), by providing a rich collection of functionals (combining forms for functions).
- Introduction of functional languages with a strong type system like ML (by Milner) and Miranda (by Turner) in the 1970s.

Some Background on SML

- Standard Meta Language (SML) was originally designed for theorem proving

  Logic for Computable Functions (Edinburgh LCF)

  Gordon, Milner, Wadsworth (1977)

Some Background on SML

- Standard Meta Language (SML) was originally designed for theorem proving

  Logic for Computable Functions (Edinburgh LCF)

  Gordon, Milner, Wadsworth (1977)

- High quality compilers, e.g. Standard ML of New Jersey and Moscow ML, based on a *formal semantics*

  Milner, Tofte, Harper, MacQueen 1990 & 1997

Some Background on SML

- Standard Meta Language (SML) was originally designed for theorem proving

  Logic for Computable Functions (Edinburgh LCF)

  Gordon, Milner, Wadsworth (1977)

- High quality compilers, e.g. Standard ML of New Jersey and Moscow ML, based on a *formal semantics*

  Milner, Tofte, Harper, MacQueen 1990 & 1997

- SML have now applications far away from its origins

  Compilers, Artificial Intelligence, Web-applications, . . .

Some Background on SML

- Standard Meta Language (SML) was originally designed for theorem proving

  Logic for Computable Functions (Edinburgh LCF)

  Gordon, Milner, Wadsworth (1977)

- High quality compilers, e.g. Standard ML of New Jersey and Moscow ML, based on a *formal semantics*

  Milner, Tofte, Harper, MacQueen 1990 & 1997

- SML have now applications far away from its origins

  Compilers, Artificial Intelligence, Web-applications, . . .

- Systems are now available on the .net platform (e.g. sml.net and F# (sml-like))

Some Background on SML

- Standard Meta Language (SML) was originally designed for theorem proving

  Logic for Computable Functions (Edinburgh LCF)

  Gordon, Milner, Wadsworth (1977)

- High quality compilers, e.g. Standard ML of New Jersey and Moscow ML, based on a *formal semantics*

  Milner, Tofte, Harper, MacQueen 1990 & 1997

- SML have now applications far away from its origins

  Compilers, Artificial Intelligence, Web-applications, . . .

- Systems are now available on the .net platform (e.g. sml.net and F# (sml-like))

- Often used to teach high-level programming concepts

Special Features

SML supports

- Functions as first class citizens

Special Features

SML supports

- Functions as first class citizens
- Structured values like lists, trees, . . .

Special Features

SML supports

- Functions as first class citizens
- Structured values like lists, trees, . . .
- Strong and flexible type discipline, including

  type inference and polymorphism

Special Features

SML supports

- Functions as first class citizens
- Structured values like lists, trees, . . .
- Strong and flexible type discipline, including

  type inference and polymorphism
- Powerful module system supporting abstract data types

Special Features

SML supports

- Functions as first class citizens
- Structured values like lists, trees, . . .
- Strong and flexible type discipline, including

  type inference and polymorphism
- Powerful module system supporting abstract data types
- Imperative programming

  assignments, loops, arrays, Input/Output, etc.

Special Features

SML supports

- Functions as first class citizens
- Structured values like lists, trees, . . .
- Strong and flexible type discipline, including

  type inference and polymorphism
- Powerful module system supporting abstract data types
- Imperative programming

  assignments, loops, arrays, Input/Output, etc.

Programming as a modelling discipline

- High-level programming, declarative programming, executable declarative specifications B, Z, VDM, RAISE
- Fast prototyping correctness, time-to-market, program designs

Overview: Part I

- The interactive environment
- Values, expressions, types, patterns
- Declarations of values and recursive functions
- Binding, environment and evaluation
- Type inference

Overview: Part I

- The interactive environment
- Values, expressions, types, patterns
- Declarations of values and recursive functions
- Binding, environment and evaluation
- Type inference

GOAL: By the end of the first part you have constructed succinct, elegant and understandable SML programs, e.g. for

- $\text{sum}(m, n) = \sum_{i=m}^{n} i$
- Fibonacci numbers ($F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$)
- Binomial coefficients $\binom{n}{k}$

The Interactive Environment

```
2*3 +4;
val it = 10 : int
```

The Interactive Environment

`2*3 +4;`	⇐ Input to the SML system
`val it = 10 : int`	⇐ Answer from the SML system

The Interactive Environment

```
2*3 +4;
val it = 10 : int
```

⇐ Input to the SML system

⇐ Answer from the SML system

- The *keyword* `val` indicates a value is computed
- The *integer* 10 is the computed value
- `int` is the *type* of the computed value
- The *identifier* `it` names the (last) computed value

The Interactive Environment

```
2*3 +4;                        ⇐ Input to the SML system
val it = 10 : int              ⇐ Answer from the SML system
```

- The *keyword* `val` indicates a value is computed
- The *integer* 10 is the computed value
- `int` is the *type* of the computed value
- The *identifier* `it` names the (last) computed value

The notion *binding* explains which entities are named by identifiers.

$\texttt{it} \mapsto 10$ reads: “`it` is bound to 10”

Value Declarations

A value declaration has the form: `val` *identifier* $=$ *expression*

```
val price = 25 * 5;          ⇐ A declaration as input
val price = 125 : int        ⇐ Answer from the SML system
```

The effect of a declaration is a binding `price` $\mapsto 125$

Value Declarations

A value declaration has the form: `val` *identifier* $=$ *expression*

```
val price = 25 * 5;          ⇐ A declaration as input
val price = 125 : int        ⇐ Answer from the SML system
```

The effect of a declaration is a binding `price` $\mapsto 125$

Bound identifiers can be used in expressions and declarations, e.g.

```
val newPrice = 2*price;
val newPrice = 250 : int

newPrice > 500;
val it = false : bool
```

Value Declarations

A value declaration has the form: `val` *identifier* = *expression*

```
val price = 25 * 5;          ⇐ A declaration as input
val price = 125 : int        ⇐ Answer from the SML system
```

The effect of a declaration is a binding $\texttt{price} \mapsto 125$

Bound identifiers can be used in expressions and declarations, e.g.

```
val newPrice = 2*price;
val newPrice = 250 : int

newPrice > 500;
val it = false : bool
```

A collection of bindings

$$\begin{bmatrix} \texttt{price} & \mapsto & 125 \\ \texttt{newPrice} & \mapsto & 250 \\ \texttt{it} & \mapsto & \text{false} \end{bmatrix}$$

is called an environment

Function Declarations 1: $\texttt{fun}\ f\ x = e$

Declaration of the circle area function:

```
fun circleArea r = Math.pi * r * r;
```

- `Math` is a program library
- `pi` is an identifier (with type `real`) for π declared in `Math`

Function Declarations 1: fun $f\,x = e$

Declaration of the circle area function:

```
fun circleArea r = Math.pi * r * r;
```

- `Math` is a program library
- `pi` is an identifier (with type `real`) for π declared in `Math`

The type is automatically inferred in the answer:

```
val circleArea = fn : real -> real
```

Function Declarations 1: fun $f\,x = e$

Declaration of the circle area function:

```
fun circleArea r = Math.pi * r * r;
```

- `Math` is a program library
- `pi` is an identifier (with type `real`) for π declared in `Math`

The type is automatically inferred in the answer:

```
val circleArea = fn : real -> real
```

Applications of the function:

```
circleArea 1.0; (* this is a comment *)
val it = 3.14159265359 : real

circleArea(3.2); (* brackets are optional *)
val it = 32.1699087728 : real
```

Recursion: $n! = 1 \cdot 2 \cdot \ldots \cdot n$, $n \geq 0$

Mathematical definition: recursion formula

$$
\begin{aligned}
0! &= 1 && \text{(i)} \\
n! &= n \cdot (n-1)!, \quad \text{for } n > 0 && \text{(ii)}
\end{aligned}
$$

Computation:

$$
\begin{aligned}
& 3! \\
= \; & 3 \cdot (3-1)! && \text{(ii)} \\
= \; & 3 \cdot 2 \cdot (2-1)! && \text{(ii)} \\
= \; & 3 \cdot 2 \cdot 1 \cdot (1-1)! && \text{(ii)} \\
= \; & 3 \cdot 2 \cdot 1 \cdot 1 && \text{(i)} \\
= \; & 6
\end{aligned}
$$

Recursive declaration: $n!$

Function declaration:

```
fun fact 0 = 1                      (* i *)
  | fact n = n * fact(n-1)          (* ii *)
val fact = fn : int -> int
```

Evaluation:

$$\begin{array}{lll} & \texttt{fact}(3) & \\ \rightsquigarrow & 3 * \texttt{fact}(3-1) & (ii) \end{array}$$

Recursive declaration: $n!$

Function declaration:

```
fun fact 0 = 1                    (* i *)
  | fact n = n * fact(n-1)        (* ii *)
val fact = fn : int -> int
```

Evaluation:

$$
\begin{array}{llll}
 & \texttt{fact}(3) & & \\
\rightsquigarrow & 3 * \texttt{fact}(3-1) & (ii) & [n \mapsto 3] \\
\rightsquigarrow & 3 * 2 * \texttt{fact}(2-1) & (ii) & [n \mapsto 2] \\
\rightsquigarrow & 3 * 2 * 1 * \texttt{fact}(1-1) & (ii) & [n \mapsto 1] \\
\rightsquigarrow & 3 * 2 * 1 * 1 & (i) & [n \mapsto 0] \\
\rightsquigarrow & 6 & &
\end{array}
$$

$e_1 \rightsquigarrow e_2$ reads: e_1 evaluates to e_2

Recursion: $x^n = x \cdot \ldots \cdot x$, n occurrences of x

Mathematical definition: recursion formula

$$
\begin{aligned}
x^0 &= 1 && (1)\\
x^n &= x \cdot x^{n-1}, \quad \text{for } n > 0 && (2)
\end{aligned}
$$

Function declaration:

```
fun power(_,0) = 1.0                    (* 1 *)
  | power(x,n) = x * power(x,n-1)       (* 2 *)
```

Patterns:

`(_,0)` matches any pair of the form $(x, 0)$.
The wildcard pattern _ matches any value.

`(x,n)` matches any pair (u, i) yielding the bindings

$$\mathtt{x} \mapsto u, \mathtt{n} \mapsto i$$

Evaluation: `power(4.0, 2)`

Function declaration:

```
fun power(_,0) = 1.0                          (* 1 *)
  | power(x,n) = x * power(x,n-1)             (* 2 *)
```

Evaluation:

	$\texttt{power}(4.0, 2)$	
$\rightsquigarrow$	$4.0 * \texttt{power}(4.0, 2 - 1)$	Clause 2, $[\texttt{x} \mapsto 4.0, \texttt{n} \mapsto 2]$
$\rightsquigarrow$	$4.0 * \texttt{power}(4.0, 1)$	
$\rightsquigarrow$	$4.0 * (4.0 * \texttt{power}(4.0, 1 - 1))$	Clause 2, $[\texttt{x} \mapsto 4.0, \texttt{n} \mapsto 1]$
$\rightsquigarrow$	$4.0 * (4.0 * \texttt{power}(4.0, 0))$	
$\rightsquigarrow$	$4.0 * (4.0 * 1)$	Clause 1
$\rightsquigarrow$	16.0	

If-then-else expressions

Form:

$$\texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2$$

Evaluation rules:

$$\texttt{if}\ \text{true}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \quad \rightsquigarrow \quad e_1$$

$$\texttt{if}\ \text{false}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \quad \rightsquigarrow \quad e_2$$

Alternative declarations:

```
fun fact n       =  if n=0 then 1
                    else n * fact(n-1);

fun power(x,n)  =  if n=0 then 1.0
                    else x * power(x,n-1);
```

If-then-else expressions

Form:

$$\texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2$$

Evaluation rules:

$$\texttt{if}\ \text{true}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \quad \rightsquigarrow \quad e_1$$
$$\texttt{if}\ \text{false}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \quad \rightsquigarrow \quad e_2$$

Alternative declarations:

```
fun fact n       =  if n=0 then 1
                    else n * fact(n-1);

fun power(x,n)  =  if n=0 then 1.0
                    else x * power(x,n-1);
```

Use of clauses and patterns often give more understandable programs

Booleans

Type name `bool`

Values `false`, `true`

Operator	Type	
`not`	`bool -> bool`	negation

```
not true = false
not false = true
```

Expressions

e_1 `andalso` e_2 "conjunction $e_1 \wedge e_2$"

e_1 `orelse` e_2 "disjunction $e_1 \vee e_2$"

— are lazily evaluated, e.g.

```
1<2 orelse 5/0 = 1
⇝ true
```

Precedence: `andalse` has higher than `orelse`

Strings

Type name `string`

Values `"abcd"`, `" "`, `""`, `"123\"321"` (escape sequence for `"`)

Operator	Type	
`size`	`string -> int`	length of string
`^`	`string*string -> string`	concatenation
`= < <= ...`	`string*string -> bool`	comparisons
`Int.toString`	`int -> string`	conversions

Examples

```
- "auto" < "car";
> val it = true : bool

- "abc"^"de";
> val it = "abcde": string
```

```
- size("abc"^"def");
> val it = 6 : int

- Int.toString(6+18);
> val it = "24" : strin
```

Types — every expression has a type $e : \tau$

Basic types:

	type name	example of values
Integers	`int`	~27, 0, 15, 21000
Reals	`real`	~27.3, 0.0, 48.21
Booleans	`bool`	true, false

Pairs:

If $e_1 : \tau_1$ and $e_2 : \tau_2$

then $(e_1, e_2) : \tau_1 * \tau_2$ pair (tuple) type constructor

Types — every expression has a type $e : \tau$

Basic types:

	type name	example of values
Integers	`int`	˜27, 0, 15, 21000
Reals	`real`	˜27.3, 0.0, 48.21
Booleans	`bool`	true, false

Pairs:

If $e_1 : \tau_1$ and $e_2 : \tau_2$

then $(e_1, e_2) : \tau_1 * \tau_2$ pair (tuple) type constructor

Functions:

if $f : \tau_1$ `->` τ_2 and $a : \tau_1$ function type constructor

then $f(a) : \tau_2$

Examples:

```
(4.0, 2):  real*int
power:  real*int -> real
power(4.0, 2):  real
```

Types — every expression has a type $e : \tau$

Basic types:

	type name	example of values
Integers	`int`	˜27, 0, 15, 21000
Reals	`real`	˜27.3, 0.0, 48.21
Booleans	`bool`	true, false

Pairs:

If $e_1 : \tau_1$ and $e_2 : \tau_2$

then $(e_1, e_2) : \tau_1 * \tau_2$ — pair (tuple) type constructor

Functions:

if $f : \tau_1$ `->` τ_2 and $a : \tau_1$ — function type constructor

then $f(a) : \tau_2$

Examples:

```
(4.0, 2):  real*int
power:  real*int -> real
power(4.0, 2):  real
```

`*` has higher precedence that `->`

Type inference: power

```
fun power(_,0) = 1.0                        (* 1 *)
  | power(x,n) = x * power(x,n-1) (* 2 *)
```

Type inference: power

```
fun power(_,0) = 1.0                   (* 1 *)
  | power(x,n) = x * power(x,n-1) (* 2 *)
```

- The type of the function must have the form: τ_1 `*` τ_2 `->` τ_3, because argument is a pair.

Type inference: power

```
fun power(_,0) = 1.0                    (* 1 *)
  | power(x,n) = x * power(x,n-1) (* 2 *)
```

- The type of the function must have the form: τ_1 `*` τ_2 `->` τ_3, because argument is a pair.
- τ_3 = `real` because `1.0:real` (Clause 1, function value.)

Type inference: power

```
fun power(_,0) = 1.0                        (* 1 *)
  | power(x,n) = x * power(x,n-1) (* 2 *)
```

- The type of the function must have the form: τ_1 `*` τ_2 `->` τ_3, because argument is a pair.
- τ_3 = `real` because `1.0:real` (Clause 1, function value.)
- τ_2 = `int` because `0:int`.

Type inference: power

```
fun power(_,0) = 1.0                    (* 1 *)
  | power(x,n) = x * power(x,n-1) (* 2 *)
```

- The type of the function must have the form: τ_1 `*` τ_2 `->` τ_3, because argument is a pair.
- τ_3 = `real` because `1.0:real` (Clause 1, function value.)
- τ_2 = `int` because `0:int`.
- `x*power(x,n-1):real`, because τ_3 = `real`.

Type inference: power

```
fun power(_,0) = 1.0                      (* 1 *)
  | power(x,n) = x * power(x,n-1) (* 2 *)
```

- The type of the function must have the form: τ_1 `*` τ_2 `->` τ_3, because argument is a pair.
- τ_3 = `real` because `1.0:real` (Clause 1, function value.)
- τ_2 = `int` because `0:int`.
- `x*power(x,n-1):real`, because τ_3 = `real`.
- multiplication can have

  `int*int -> int` or `real*real -> real`

  as types, but no “mixture” of `int` and `real`

Type inference: power

```
fun power(_,0) = 1.0                      (* 1 *)
  | power(x,n) = x * power(x,n-1) (* 2 *)
```

- The type of the function must have the form: τ_1 `*` τ_2 `->` τ_3, because argument is a pair.
- τ_3 = `real` because `1.0:real` (Clause 1, function value.)
- τ_2 = `int` because `0:int`.
- `x*power(x,n-1):real`, because τ_3 = `real`.
- multiplication can have

  `int*int -> int` or `real*real -> real`

  as types, but no “mixture” of `int` and `real`
- Therefore `x:real` and τ_1=`real`.

Type inference: power

```
fun power(_,0) = 1.0                    (* 1 *)
  | power(x,n) = x * power(x,n-1) (* 2 *)
```

- The type of the function must have the form: τ_1 `*` τ_2 `->` τ_3, because argument is a pair.
- τ_3 = `real` because `1.0:real` (Clause 1, function value.)
- τ_2 = `int` because `0:int`.
- `x*power(x,n-1):real`, because τ_3 = `real`.
- multiplication can have

  `int*int -> int` or `real*real -> real`

  as types, but no “mixture” of `int` and `real`
- Therefore `x:real` and τ_1=`real`.

The SML system determines the type `real*int -> real`

Summary

- The interactive environment
- Values, expressions, types, patterns
- Declarations of values and recursive functions
- Binding, environment and evaluation
- Type inference

Breath first round through many concepts aiming at program construction from the first day.

We will go deaper into each of the concepts later in the course.