Introduction to SML

Getting Started

Michael R. Hansen

m rh@imm.dtu.dk

Informatics and Mathematical Modelling

Technical University of Denmark
Standard Meta Language (SML) was originally designed for theorem proving

Logic for Computable Functions (Edinburgh LCF)

Gordon, Milner, Wadsworth (1977)
Background

- 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
Background

- 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 many applications far away from its origins
  Compilers, Artificial Intelligence, Web-applications, ...
Background

• 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 many applications far away from its origins
  Compilers, Artificial Intelligence, Web-applications, . . .

• Used to teach functional program design and programming style
Background

- 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 may applications far away from its origins
  Compilers, Artificial Intelligence, Web-applications, ...

- Used to teach functional program design and programming style
  Also useful when programming using “non-functional” languages
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**
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 specifications VDM, RAISE
• Fast prototyping correctness, time-to-market, program designs
Overview

- 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 day you have constructed succinct, elegant and understandable SML programs, e.g. for

\[
\text{sum}(m; n) = \sum_{i=0}^{n} m_i = m_i
\]

Fibonacci numbers (\(F_0 = 0; F_1 = 1; F_n = F_{n-1} + F_{n-2}\))

Binomial coefficients \(\binom{n}{k}\)!
Overview

- 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 day 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 \( \binom{n}{k} \)
2*3 +4;
val it = 10 : int
2*3 +4;  
val it = 10 : int  
\(\Leftarrow\) Input to the SML system

\(\Leftarrow\) Answer from the SML system
The Interactive Environment

\(2 \times 3 + 4;\) \hspace{1cm} \text{Input to the SML system}

\text{val it = 10 : int} \hspace{1cm} \text{Answer from the SML system}

- The \textit{keyword} \texttt{val} indicates a value is computed
- The \textit{integer} 10 is the computed value
- \texttt{int} is the \textit{type} of the computed value
- The \textit{identifier} \texttt{it} names the (last) computed value
The Interactive Environment

\[ 2 \times 3 + 4; \]
\[ \text{val } \texttt{it} = 10 : \texttt{int} \]

\[ \Leftarrow \text{Input to the SML system} \]
\[ \Leftarrow \text{Answer from the SML system} \]

- The \textit{keyword} \texttt{val} indicates a value is computed
- The \textit{integer} 10 is the computed value
- \texttt{int} is the \textit{type} of the computed value
- The \textit{identifier} \texttt{it} names the (last) computed value

The notion \textit{binding} explains which entities are named by identifiers.

\[ \texttt{it} \mapsto 10 \]
reads: “\texttt{it} is bound to 10”
Value Declarations

A value declaration has the form: \texttt{val identifier = expression}

\begin{align*}
\texttt{val price} & = 25 \times 5; & \iff & \text{A declaration as input} \\
\texttt{val price} & = 125 : \texttt{int} & \iff & \text{Answer from the SML system}
\end{align*}

The effect of a declaration is a binding

\[
\begin{array}{c}
\text{price} \mapsto 125
\end{array}
\]
Value Declarations

A value declaration has the form: \texttt{val identifier = expression}

\begin{verbatim}
val price = 25 * 5;
\end{verbatim} \hfill A declaration as input

\begin{verbatim}
val price = 125 : int
\end{verbatim} \hfill Answer from the SML system

The effect of a declaration is a binding

\begin{verbatim}
price \rightarrow 125
\end{verbatim}

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

\begin{verbatim}
val newPrice = 2*price;
val newPrice = 250 : int
\end{verbatim}

\begin{verbatim}
newPrice > 500;
val it = false : bool
\end{verbatim}
Value Declarations

A value declaration has the form: \texttt{val identifier = expression}

\begin{align*}
\texttt{val price} & \;= \;25 \ast 5; \quad \Leftarrow \text{A declaration as input} \\
\texttt{val price} & \;= \;125 : \texttt{int} \quad \Leftarrow \text{Answer from the SML system}
\end{align*}

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

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

\begin{align*}
\texttt{val newPrice} & \;= \;2 \ast \texttt{price}; \\
\texttt{val newPrice} & \;= \;250 : \texttt{int} \\
\texttt{newPrice} & \;> \;500; \\
\texttt{val it} & \;= \;\texttt{false} : \texttt{bool}
\end{align*}

\textbf{A collection of bindings}

\[
\begin{bmatrix}
\texttt{price} & \mapsto & 125 \\
\texttt{newPrice} & \mapsto & 250 \\
\texttt{it} & \mapsto & \texttt{false}
\end{bmatrix}
\]

is called an \textbf{environment}
Function Declarations 1: \( \text{fun } \ f \ x \ = \ e \)

Declaration of the circle area function:

\[
\text{fun circleArea } r = \text{Math.pi} * r * r;
\]

- \textbf{Math} is a program library
- \textit{pi} is an identifier (with type real) for \( \pi \) declared in Math
Function Declarations 1: \( \text{fun } f x = e \)

Declaration of the circle area function:
\[
\text{fun circleArea } r = \text{Math.pi } \ast r \ast r;
\]

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

The type is automatically inferred in the answer:
\[
\text{val circleArea } = \text{fn } : \text{real } \rightarrow \text{real}
\]
Declaration of the circle area function:

\[
\text{fun circleArea } r = \text{Math.pi} * r * r;
\]

- Math is a program library
- \text{pi} is an identifier (with type real) for \(\pi\) declared in Math

The type is automatically inferred in the answer:

\[
\text{val circleArea} = \text{fn : real} \to \text{real}
\]

Applications of the function:

\[
\text{circleArea } 1.0; (* \text{this is a comment} *)
\]
\[
\text{val it} = 3.14159265359 : \text{real}
\]

\[
\text{circleArea} (3.2); (* \text{brackets are optional} *)
\]
\[
\text{val it} = 32.1699087728 : \text{real}
\]
Recursion: \( n! = 1 \cdot 2 \cdot \ldots \cdot n, \ n \geq 0 \)

Mathematical definition:

\[
egin{align*}
0! &= 1 \\
n! &= n \cdot (n - 1)!, \quad \text{for } n > 0
\end{align*}
\]

Computation:

\[
egin{align*}
3! &= 3 \cdot (3 - 1)! \quad \text{(ii)} \\
&= 3 \cdot 2 \cdot (2 - 1)! \quad \text{(ii)} \\
&= 3 \cdot 2 \cdot 1 \cdot (1 - 1)! \quad \text{(ii)} \\
&= 3 \cdot 2 \cdot 1 \cdot 1 \quad \text{(i)} \\
&= 6
\end{align*}
\]
Recursive declaration: $n!$

Function declaration:

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

Evaluation:

```
fact(3)
\mapsto 3 * fact(3 - 1) (ii)
\mapsto 3 * 2 * fact(2 - 1) (ii)
\mapsto 3 * 2 * 1 * fact(1 - 1) (ii)
\mapsto 3 * 2 * 1 * 1 (i)
\mapsto 6
```

\[e_1 \mapsto e_2\] reads: \(e_1\) evaluates to \(e_2\)
Recursive declaration: \( n! \)

Function declaration:

\[
\text{fun fact } 0 = 1 \quad (* \text{i } *)
\]

\[
\mid \text{fact } n = n \times \text{fact}(n-1) \quad (* \text{ii } *)
\]

\[
\text{val fact } = \text{fn : int } \to \text{ int}
\]

Evaluation:

\[
\text{fact}(3)
\]

\[
\leadsto 3 \times \text{fact}(3 - 1) \quad \text{(ii) } [n \mapsto 3]
\]

\[
\leadsto 3 \times 2 \times \text{fact}(2 - 1) \quad \text{(ii) } [n \mapsto 2]
\]

\[
\leadsto 3 \times 2 \times 1 \times \text{fact}(1 - 1) \quad \text{(ii) } [n \mapsto 1]
\]

\[
\leadsto 3 \times 2 \times 1 \times 1 \quad \text{(i) } [n \mapsto 0]
\]

\[
\leadsto 6
\]
Recursion: $x^n = x \cdot \ldots \cdot x$, $n$ occurrences of $x$

Mathematical definition:

recursion formula

\[
\begin{align*}
  x^0 &= 1 \\
  x^n &= x \cdot x^{n-1}, \text{ for } n > 0
\end{align*}
\]

Function declaration:

\[
\begin{align*}
  \text{fun power(_,0) = 1.0 (* 1 *)} \\
  | \text{power}(x,n) = x * \text{power}(x,n-1) (* 2 *)
\end{align*}
\]

Patterns:

(\_, \text{0}) \text{ matches any pair of the form } (x, 0).

The wildcard pattern \_ matches any value.

(x, n) \text{ matches any pair } (u, i) \text{ yielding the bindings }

\[
x \mapsto u, n \mapsto i
\]
Evaluation: \texttt{power}(4.0, 2)

Function declaration:

\begin{align*}
    \text{fun power}(\_, 0) &= 1.0 \quad (* 1 *) \\
    | \quad \text{power}(x, n) &= x \times \text{power}(x, n-1) \quad (* 2 *)
\end{align*}

Evaluation:

\begin{align*}
    \text{power}(4.0, 2) \\
    \leadsto 4.0 \times \text{power}(4.0, 2 - 1) & \quad \text{Clause 2, } [x \mapsto 4.0, n \mapsto 2] \\
    \leadsto 4.0 \times \text{power}(4.0, 1) \\
    \leadsto 4.0 \times (4.0 \times \text{power}(4.0, 1 - 1)) & \quad \text{Clause 2, } [x \mapsto 4.0, n \mapsto 1] \\
    \leadsto 4.0 \times (4.0 \times \text{power}(4.0, 0)) \\
    \leadsto 4.0 \times (4.0 \times 1) & \quad \text{Clause 1} \\
    \leadsto 16.0
\end{align*}
If-then-else expressions

Form:

\[ \text{if } b \text{ then } e_1 \text{ else } e_2 \]

Evaluation rules:

\[ \text{if true then } e_1 \text{ else } e_2 \rightsquigarrow e_1 \]
\[ \text{if false then } e_1 \text{ else } e_2 \rightsquigarrow e_2 \]

Alternative declarations:

\begin{align*}
\text{fun fact } n & = \text{ if } n=0 \text{ then 1 } \\
& \quad \text{else } n \times \text{ fact}(n-1) ; \\
\text{fun power}(x, n) & = \text{ if } n=0 \text{ then 1.0 } \\
& \quad \text{else } x \times \text{ power}(x, n-1) ;
\end{align*}
If-then-else expressions

Form:

\[ \text{if } b \text{ then } e_1 \text{ else } e_2 \]

Evaluation rules:

\[ \text{if } \text{true} \text{ then } e_1 \text{ else } e_2 \mapsto e_1 \]
\[ \text{if } \text{false} \text{ then } e_1 \text{ else } e_2 \mapsto e_2 \]

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

<table>
<thead>
<tr>
<th>Basic types:</th>
<th>type name</th>
<th>example of values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integers</td>
<td>int</td>
<td>$\sim 27, 0, 15, 21000$</td>
</tr>
<tr>
<td>Reals</td>
<td>real</td>
<td>$\sim 27.3, 0.0, 48.21$</td>
</tr>
<tr>
<td>Booleans</td>
<td>bool</td>
<td>true, false</td>
</tr>
</tbody>
</table>

Pairs: If $e_1 : \tau_1$ and $e_2 : \tau_2$ then $(e_1, e_2) : \tau_1 \times \tau_2$ pair (tuple) type constructor

Examples:

$(4.0, 2) : \text{real} \times \text{int}$
Types — every expression has a type \( e : \tau \)

<table>
<thead>
<tr>
<th>type name</th>
<th>example of values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integers</td>
<td>int</td>
</tr>
<tr>
<td>Reals</td>
<td>real</td>
</tr>
<tr>
<td>Booleans</td>
<td>bool</td>
</tr>
</tbody>
</table>

Pairs: If \( e_1 : \tau_1 \) and \( e_2 : \tau_2 \) then \( (e_1, e_2) : \tau_1 \times \tau_2 \)  

Function type constructor:
If \( f : \tau_1 \rightarrow \tau_2 \) and \( a : \tau_1 \) then \( f(a) : \tau_2 \)

Examples:
- \((4.0, 2) : \text{real} \times \text{int}\)
- \text{power} : \text{real} \times \text{int} \rightarrow \text{real}
- \text{power}(4.0, 2) : \text{real}
Types — every expression has a type $e : \tau$

<table>
<thead>
<tr>
<th>Basic types:</th>
<th>type name</th>
<th>example of values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integers</td>
<td>int</td>
<td>$-27, 0, 15, 21000$</td>
</tr>
<tr>
<td>Reals</td>
<td>real</td>
<td>$-27.3, 0.0, 48.21$</td>
</tr>
<tr>
<td>Booleans</td>
<td>bool</td>
<td>true, false</td>
</tr>
</tbody>
</table>

Pairs: If $e_1 : \tau_1$ and $e_2 : \tau_2$ then $(e_1, e_2) : \tau_1 \times \tau_2$

Functions: if $f : \tau_1 \to \tau_2$ and $a : \tau_1$ then $f(a) : \tau_2$

Examples:

$\text{(4.0, 2)}: \text{real} \times \text{int}$

$\text{power}: \text{real} \times \text{int} \to \text{real}$  
* has higher precedence than $\to$

$\text{power}(4.0, 2): \text{real}$
Type inference: power

fun power(_,0) = 1.0 (* 1 *)
| power(x,n) = x * power(x,n-1) (* 2 *)
Type inference: \texttt{power}

\begin{verbatim}
fun power(_,0) = 1.0 (* 1 *)
| power(x,n) = x * power(x,n-1) (* 2 *)
\end{verbatim}

- The type of the function must have the form: $\tau_1 \times \tau_2 \rightarrow \tau_3$, because argument is a pair.
fun power(_,0) = 1.0 (* 1 *)
| power(x,n) = x * power(x,n-1) (* 2 *)

• The type of the function must have the form: $\tau_1 \times \tau_2 \rightarrow \tau_3$, because argument is a pair.

• $\tau_3 = \text{real because } 1.0: \text{real}$ (Clause 1, function value.)
fun power(_,0) = 1.0  (* 1 *)
   | power(x,n) = x * power(x,n-1)  (* 2 *)

• The type of the function must have the form: $\tau_1 \times \tau_2 \rightarrow \tau_3$, because argument is a pair.

• $\tau_3 = \text{real}$ because $1.0: \text{real}$  (Clause 1, function value.)

• $\tau_2 = \text{int}$ because $0: \text{int}$. 
fun power(_,0) = 1.0
   | power(x,n) = x * power(x,n-1) (* 2 *)

• The type of the function must have the form: \( \tau_1 \times \tau_2 \rightarrow \tau_3 \), because argument is a pair.

• \( \tau_3 = \text{real} \) because \( 1.0 : \text{real} \) (Clause 1, function value.)

• \( \tau_2 = \text{int} \) because \( 0 : \text{int} \).

• \( x \times \text{power}(x,n-1) : \text{real} \), because \( \tau_3 = \text{real} \).
fun power(_,0) = 1.0 (* 1 *)
    | power(x,n) = x * power(x,n-1) (* 2 *)

• The type of the function must have the form: \( \tau_1 \times \tau_2 \rightarrow \tau_3 \), because argument is a pair.

• \( \tau_3 = \text{real} \) because \( 1.0: \text{real} \) (Clause 1, function value.)

• \( \tau_2 = \text{int} \) because \( 0: \text{int} \).

• \( x \times \text{power}(x,n-1): \text{real} \), because \( \tau_3 = \text{real} \).

• Multiplication can have

  \[ \text{int} \times \text{int} \rightarrow \text{int} \text{ or } \text{real} \times \text{real} \rightarrow \text{real} \]

  as types, but no “mixture” of \( \text{int} \) and \( \text{real} \).
fun power(_,0) = 1.0 (* 1 *)
      | power(x,n) = x * power(x,n-1) (* 2 *)

• The type of the function must have the form: \( \tau_1 \times \tau_2 \rightarrow \tau_3 \), because argument is a pair.
• \( \tau_3 = \text{real} \) because \( 1.0 : \text{real} \) (Clause 1, function value.)
• \( \tau_2 = \text{int} \) because \( 0 : \text{int} \).
• \( x \times \text{power}(x,n-1) : \text{real} \), because \( \tau_3 = \text{real} \).
• multiplication can have
  \( \text{int} \times \text{int} \rightarrow \text{int} \) or \( \text{real} \times \text{real} \rightarrow \text{real} \)
  as types, but no “mixture” of \( \text{int} \) and \( \text{real} \)
• Therefore \( x : \text{real} \) and \( \tau_1 = \text{real} \).
fun power(_,0) = 1.0 (* 1 *)
    | power(x,n) = x * power(x,n-1) (* 2 *)

• The type of the function must have the form: \( \tau_1 \times \tau_2 \rightarrow \tau_3 \), because argument is a pair.
• \( \tau_3 = \text{real} \) because \( 1.0:\text{real} \) (Clause 1, function value.)
• \( \tau_2 = \text{int} \) because \( 0:\text{int} \).
• \( x \times \text{power}(x,n-1):\text{real} \), because \( \tau_3 = \text{real} \).
• multiplication can have
  \[ \text{int} \times \text{int} \rightarrow \text{int} \text{ or } \text{real} \times \text{real} \rightarrow \text{real} \]
  as types, but no “mixture” of \( \text{int} \) and \( \text{real} \)
• Therefore \( x:\text{real} \) and \( \tau_1 = \text{real} \).

The SML system determines the type \( \text{real} \times \text{int} \rightarrow \text{real} \)
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 deeper into each of the concepts later in the course.