02157 Functional Programming

Interpreters for two simple languages
– including exercises

Michael R. Hansen

DTU Compute
Department of Applied Mathematics and Computer Science

Overview

- Miscellaneous
  - On type declarations — Type abbreviations and ”real” types
- Finite trees: Two examples
  - An interpreter for a simple expression language
  - An interpreter for a simple while-language

On type declarations (1)

A tagged-value type like `T1`:

```
type T1 = | A of int | B of bool
```

is a real new type.

On type declarations (1)

A tagged-value type like `T1`:

```
type T1 = | A of int | B of bool
```

is a real new type.

This type is different from the type `T2`:

```
type T2 = | A of int | B of bool;;

let v1 = T1.A(2);;
let v2 = T2.A(2);;
```

On type declarations (1)

A tagged-value type like `T1`:

```
type T1 = | A of int | B of bool
```

is a real new type.

This type is different from the type `T2`:

```
type T2 = | A of int | B of bool;;

let v1 = T1.A(2);;
let v2 = T2.A(2);;

v1=v2;;
... error ... : This expression was expected to have type
   T1
but here has type
   T2
```

On type declarations (1)

A tagged-value type like `T1`:

```
type T1 = | A of int | B of bool
```

is a real new type.

This type is different from the type `T2`:

```
type T2 = | A of int | B of bool;;

let v1 = T1.A(2);;
let v2 = T2.A(2);;

v1=v2;;
... error ... : This expression was expected to have type
   T1
but here has type
   T2
```

- Values of type `T1` and `T2` cannot be compared
  — the types are, in fact, different.
- A similar observation applies for records.

On type declarations (2)

The other kinds of type declarations we have considered so far define type abbreviations. For example, the types `TA1` and `TA2`:

```
type TA1 = int * bool;;
type TA2 = int * bool;;
```

are identical and just shorthands for `int * bool`.

On type declarations (2)

The other kinds of type declarations we have considered so far define type abbreviations. For example, the types `TA1` and `TA2`:

```
type TA1 = int * bool;;
type TA2 = int * bool;;
```

are identical and just shorthands for `int * bool`.

For example:

```
let v1 = (1,true):TA1;;
val v1 : TA1 = (1, true)
let v2 = (1,true):TA2;;
val v2 : TA2 = (1, true)
let v3 = (1,true):int*bool;;
val v3 : int * bool = (1, true)

v1=v2;;
val it : bool = true

v2=v3;;
val it : bool = true
```

On type declarations (3)

Type declarations, including type abbreviations, often have a pragmatic role. For example, succinct way of presenting a model:

```
type CourseNo   = int
type Title      = string
type ECTS       = int
type CourseDesc = Title * ECTS

type CourseBase = Map<CourseNo, CourseDesc>

type Mandatory   = Set<CourseNo>
type Optional    = Set<CourseNo>
type CourseGroup = Mandatory * Optional
```

On type declarations (3)

Type declarations, including type abbreviations, often have a pragmatic role. For example, succinct way of presenting a model:

```
type CourseNo    = int
type Title       = string
type ECTS        = int
type CourseDesc = Title * ECTS

type CourseBase = Map<CourseNo, CourseDesc>

type Mandatory    = Set<CourseNo>
type Optional     = Set<CourseNo>
type CourseGroup = Mandatory * Optional
```

or the purpose of a function like

```
isValidCourseGroup: CourseGroup -> CourseBase -> bool
```

On type declarations (3)

Type declarations, including type abbreviations, often have a pragmatic role. For example, succinct way of presenting a model:

```
type CourseNo   = int
type Title      = string
type ECTS       = int
type CourseDesc = Title * ECTS

type CourseBase = Map<CourseNo, CourseDesc>

type Mandatory   = Set<CourseNo>
type Optional    = Set<CourseNo>
type CourseGroup = Mandatory * Optional
```

or the purpose of a function like

```
isValidCourseGroup: CourseGroup -> CourseBase -> bool
```

The expanded type for this function is NOT “enriching”:

```
(Set<int>*Set<int>) -> Map<int,string*int> -> bool
```

Interpreters for two simple languages — Purpose

To show the power of a functional programming language, we present a prototype for interpreters for a simple expression language with local declarations and a simple WHILE language.

Interpreters for two simple languages — Purpose

To show the power of a functional programming language, we present a prototype for interpreters for a simple expression language with local declarations and a simple WHILE language.

- Concrete syntax: defined by a contextfree grammar

Interpreters for two simple languages — Purpose

To show the power of a functional programming language, we present a prototype for interpreters for a simple expression language with local declarations and a simple WHILE language.

- Concrete syntax: defined by a contextfree grammar
- Abstract syntax (parse trees): defined by algebraic datatypes

Interpreters for two simple languages — Purpose

To show the power of a functional programming language, we present a prototype for interpreters for a simple expression language with local declarations and a simple WHILE language.

- Concrete syntax: defined by a contextfree grammar
- Abstract syntax (parse trees): defined by algebraic datatypes
- Semantics, i.e. meaning of programs: inductively defined following the structure of the abstract syntax

succinct programs, fast prototyping

Interpreters for two simple languages — Purpose

To show the power of a functional programming language, we present a prototype for interpreters for a simple expression language with local declarations and a simple WHILE language.

- Concrete syntax: defined by a contextfree grammar
- Abstract syntax (parse trees): defined by algebraic datatypes
- Semantics, i.e. meaning of programs: inductively defined following the structure of the abstract syntax

succinct programs, fast prototyping

The interpreter for the simple expression language is a higher-order function:

$$eval : Program \rightarrow Environment \rightarrow Value$$

The interpreter for a simple imperative programming language is a higher-order function:

$$I : Program \rightarrow State \rightarrow State$$

Expressions with local declarations

Concrete syntax:

```
a * (-3 + (let x = 5 in x + a))
```

Expressions with local declarations

Concrete syntax:

```
a * (-3 + (let x = 5 in x + a))
```

The abstract syntax is defined by an algebraic datatype:

```
type ExprTree = | Const of int
                | Ident of string
                | Minus of ExprTree
                | Sum   of ExprTree * ExprTree
                | Diff  of ExprTree * ExprTree
                | Prod  of ExprTree * ExprTree
                | Let of string * ExprTree * ExprTree;;
```

Expressions with local declarations

Concrete syntax:

```
a * (-3 + (let x = 5 in x + a))
```

The abstract syntax is defined by an algebraic datatype:

```
type ExprTree = | Const of int
                | Ident of string
                | Minus of ExprTree
                | Sum   of ExprTree * ExprTree
                | Diff  of ExprTree * ExprTree
                | Prod  of ExprTree * ExprTree
                | Let of string * ExprTree * ExprTree;;
```

Example:

```
let et =
  Prod(Ident "a",
       Sum(Minus (Const 3),
           Let("x", Const 5, Sum(Ident "x", Ident "a"))));;
```

Evaluation in Environments

An *environment* contains *bindings* of identifiers to values.

Evaluation in Environments

An *environment* contains *bindings* of identifiers to values.

A tree `Let(`s`,`t_1`,`t_2`)` is evaluated in an environment *env*:

1. Evaluate t_1 to value v_1 in environment *env*.
2. Evaluate t_2 in *env* extended with the binding of s to v_1.

Evaluation in Environments

An *environment* contains *bindings* of identifiers to values.

A tree `Let(`s, t_1, t_2`)` is evaluated in an environment *env*:

1. Evaluate t_1 to value v_1 in environment *env*.
2. Evaluate t_2 in *env* extended with the binding of s to v_1.

An evaluation function

```
eval: ExprTree -> Map<string,int> -> int
```

is defined as follows:

```
let rec eval t env =
    match t with
    | Const n      -> n
    | Ident s      -> Map.find s env
    | Minus t      -> - (eval t env)
    | Sum(t1,t2)   -> eval t1 env + eval t2 env
    | Diff(t1,t2)  -> eval t1 env - eval t2 env
    | Prod(t1,t2)  -> eval t1 env * eval t2 env
    | Let(s,t1,t2) -> let v1   = eval t1 env
                      let env1 = Map.add s v1 env
                      eval t2 env1;;
```

Example

Concrete syntax:

```
a * (-3 + (let x = 5 in x + a))
```

Example

Concrete syntax:

```
a * (-3 + (let x = 5 in x + a))
```

```
let et =
  Prod(Ident "a",
       Sum(Minus (Const 3),
           Let("x", Const 5, Sum(Ident "x", Ident "a"))));;
```

Example

Concrete syntax:

```
a * (-3 + (let x = 5 in x + a))
```

```
let et =
  Prod(Ident "a",
       Sum(Minus (Const 3),
           Let("x", Const 5, Sum(Ident "x", Ident "a"))));;
```

```
let env = Map.add "a" -7 Map.empty;;
eval et env;;
val it : int = 35
```

Example: Imperative Factorial program

An example of concrete syntax for a factorial program:

```
y:=1 ;
while !(x=0)
do (y:= y*x;x:=x-1)
```

Example: Imperative Factorial program

An example of concrete syntax for a factorial program:

```
y:=1 ;
while !(x=0)
do (y:= y*x;x:=x-1)
```

Typical ingredients

- Arithmetical expressions

Example: Imperative Factorial program

An example of concrete syntax for a factorial program:

```
y:=1 ;
while !(x=0)
do (y:= y*x;x:=x-1)
```

Typical ingredients

- Arithmetical expressions
- Boolean expressions

Example: Imperative Factorial program

An example of concrete syntax for a factorial program:

```
y:=1 ;
while !(x=0)
do (y:= y⋆x;x:=x-1)
```

Typical ingredients

- Arithmetical expressions
- Boolean expressions
- Statements (assignments, sequential composition, loops, . . .

Arithmetic Expressions

- The declaration of the abstract syntax for Arithmetical Expressions

```
type AExp =        (* Arithmetical expressions *)
   | N   of int                (* numbers        *)
   | V   of string             (* variables      *)
   | Add of AExp * AExp        (* addition       *)
   | Mul of AExp * AExp        (* multiplication *)
   | Sub of AExp * AExp;;      (* subtraction    *)
```

Arithmetic Expressions

- The declaration of the abstract syntax for Arithmetical Expressions

```
type AExp =        (* Arithmetical expressions *)
   | N   of int                (* numbers          *)
   | V   of string             (* variables        *)
   | Add of AExp * AExp        (* addition         *)
   | Mul of AExp * AExp        (* multiplication   *)
   | Sub of AExp * AExp;;      (* subtraction      *)
```

You do not need parenthesis, precedence rules, ect for the abstract syntax — you work directly on trees.

Semantics of Arithmetic Expressions

- A state maps variables to integers

```
type State = Map<string,int>;;
```

Semantics of Arithmetic Expressions

- A state maps variables to integers

```
type State = Map<string,int>;;
```

- The meaning of an expression is a function:

```
A: AExp -> State -> int
```

Semantics of Arithmetic Expressions

- A state maps variables to integers

```
type State = Map<string,int>;;
```

- The meaning of an expression is a function:

```
A: AExp -> State -> int
```

  defined inductively on the structure of arithmetic expressions

```
let rec A a s       =
   match a with
    | N n          -> n
    | V x          -> Map.find x s
    | Add(a1, a2)  -> A a1 s + A a2 s
    | Mul(a1, a2)  -> A a1 s * A a2 s
    | Sub(a1, a2)  -> A a1 s - A a2 s;;
```

Boolean Expressions

- Abstract syntax

```
type BExp =              (* Boolean expressions *)
    | TT                         (* true             *)
    | FF                         (* false            *)
    | Eq of ....                 (* equality         *)
    | Lt of ....                 (* less than        *)
    | Neg of ....                (* negation         *)
    | Con of ....         ;;     (* conjunction      *)
```

Boolean Expressions

- Abstract syntax

```
type BExp =                  (* Boolean expressions *)
   | TT                             (* true             *)
   | FF                             (* false            *)
   | Eq of ....                     (* equality         *)
   | Lt of ....                     (* less than        *)
   | Neg of ....                    (* negation         *)
   | Con of ....           ;;       (* conjunction      *)
```

- Semantics B : BExp $\rightarrow$ State $\rightarrow$ bool

```
let rec B b s =
   match b with
   | TT       -> true
   | ....
```

Statements: Abstract Syntax

```
type Stm  =                          (* statements             *)
   | Ass of string * AExp            (* assignment             *)
   | Skip
   | Seq  of Stm * Stm               (* sequential composition *)
   | ITE   of BExp * Stm * Stm       (* if-then-else           *)
   | While of BExp * Stm;;           (* while                  *)
```

Statements: Abstract Syntax

```
type Stm  =                              (* statements              *)
   | Ass of string * AExp                (* assignment              *)
   | Skip
   | Seq  of Stm * Stm                   (* sequential composition *)
   | ITE   of BExp * Stm * Stm           (* if-then-else            *)
   | While of BExp * Stm;;               (* while                   *)
```

Example of concrete syntax:

```
y:=1 ; while not(x=0) do (y:= y*x ; x:=x-1)
```

Abstract syntax ?

Update of states

An imperative program performs a sequence of state updates.

- The expression

  $$\texttt{update}\ y\ v\ s$$

  is the state that is as s except that y is mapped to v.

Update of states

An imperative program performs a sequence of state updates.

- The expression

  `update` $y\ v\ s$

  is the state that is as s except that y is mapped to v.
  Mathematically:

$$(update\ y\ v\ s)(x) = \begin{cases} v & \text{if } x = y \\ s(x) & \text{if } x \neq y \end{cases}$$

Update of states

An imperative program performs a sequence of state updates.

- The expression

  $$\texttt{update}\ y\ v\ s$$

  is the state that is as s except that y is mapped to v.
  Mathematically:

  $$(\mathit{update}\ y\ v\ s)(x) = \begin{cases} v & \text{if } x = y \\ s(x) & \text{if } x \neq y \end{cases}$$

- Update is a higher-order function with the declaration:

  ```
  let update x v s = Map.add x v s;;
  ```

- Type?

Interpreter for Statements

- The meaning of statements is a function

$$I : Stm \rightarrow State \rightarrow State$$

that is defined by induction on the structure of statements:

```
let rec I stm s =
    match stm with
    | Ass(x,a)          -> update x ( ... ) s
    | Skip              -> ...
    | Seq(stm1, stm2)   -> ...
    | ITE(b,stm1,stm2)  -> ...
    | While(b, stm)     -> ... ;;
```

Example: Factorial function

```
(*
      y:=1 ; while !(x=0) do (y:= y*x;x:=x-1)
*)
```

Example: Factorial function

```
(*
      y:=1 ; while !(x=0) do (y:= y*x;x:=x-1)
*)


let fac = Seq(Ass("y", N 1),
              While(Neg(Eq(V "x", N 0)),
                    Seq(Ass("y", Mul(V "x", V "y")) ,
                        Ass("x", Sub(V "x", N 1)) )));;
```

Example: Factorial function

```
(*
      y:=1 ; while !(x=0) do (y:= y*x;x:=x-1)
*)


let fac = Seq(Ass("y", N 1),
              While(Neg(Eq(V "x", N 0)),
                    Seq(Ass("y", Mul(V "x", V "y")) ,
                        Ass("x", Sub(V "x", N 1)) )));;


(* Define an initial state                                  *)
let s0 = Map.ofList [("x",4)];;
val s0 : Map<string,int> = map [("x", 4)]
```

Example: Factorial function

```
(*
      y:=1 ; while !(x=0) do (y:= y*x;x:=x-1)
*)

let fac = Seq(Ass("y", N 1),
              While(Neg(Eq(V "x", N 0)),
                    Seq(Ass("y", Mul(V "x", V "y")) ,
                        Ass("x", Sub(V "x", N 1)) )));;

(* Define an initial state                                    *)
let s0 = Map.ofList [("x",4)];;
val s0 : Map<string,int> = map [("x", 4)]

(* Interpret the program                                      *)
let s1 = I fac s0;;
val s1 : Map<string,int> = map [("x", 1); ("y", 24)]
```

Exercises

- Complete the program skeleton for the interpreter, and try some examples.
- Extend the abstract syntax and the interpreter with if-then and repeat-until statements.
- Suppose that an expression of the form $inc(x)$ is added. It adds one to the value of x in the current state, and the value of the expression is this new value of x.

  How would you refine the interpreter to cope with this construct?

  - Analyse the problem and state the types for the refined interpretation functions