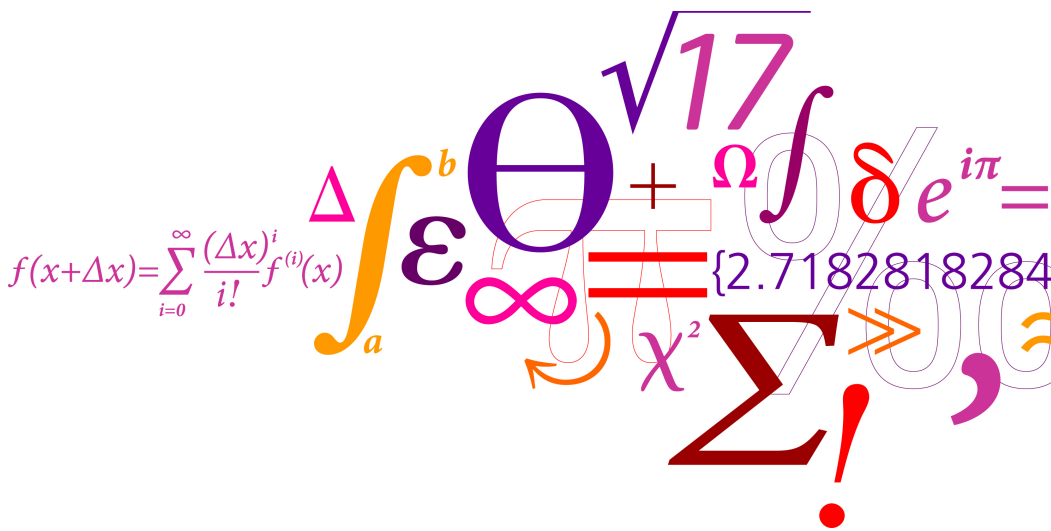
02157 Functional Programming

Finite trees (II)

– Interpreters for two simple languages

Michael R. Hansen



DTU Compute
Department of Applied Mathematics and Computer Science

Overview

Finite trees: Two examples

- An interpreter for a simple expression language
- An interpreter for a simple while-language

Finite trees (II)

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:

$$eval : Program \to Environment \to Value$$

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

$$I : Program \to State \to State$$

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

- A type 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

- A type 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, etc.
in 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 $\mathsf{B} : \mathsf{BExp} \to \mathsf{State} \to \mathsf{bool}$

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

Statements: Abstract Syntax

```
type Stm  =                          (* statements                *)
    | Ass of string * AExp           (* assignment                *)
    | Seq  of Stm list               (* 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                *)
    | Seq  of Stm list               (* 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*.

- Update is a synonym for `Map.add`:

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

Interpreter for Statements

- The meaning of statements is a function

$$I : Stm \to State \to State$$

that is defined by induction on the structure of statements:

```
let rec I stm s =
    match stm with
    | Ass(x,a)          -> update x ( ... ) s
    | Seq stms          -> ...
    | 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)]
```

Interpreter will be available on Learn.

- You may add the statements skip, if-then and repeat-until.
- 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