Functional Programming

WS 2012/13

Harald Zankl (VO)
Cezary Kaliszyk (PS) Thomas Sternagel (PS)

Computational Logic
Institute of Computer Science
University of Innsbruck

week 12

Type Checking

- prove that expression e has a type τ w.r.t. environment E
- formally: $E \vdash e : \tau$
- use the inference rules of $\mathcal{C}$ to do so

Type Inference

- find most general type $\alpha_0\sigma$ for expression e w.r.t. environment E
- formally: $E \rhd e : \alpha_0$
- task is split into two parts:
  1. transform given type inference problem into unification problem
  2. solve the unification problem (result is substitution σ)

This Week

Practice I

OCaml introduction, lists, strings, trees

Theory I

lambda-calculus, evaluation strategies, induction, reasoning about functional programs

Practice II

efficiency, tail-recursion, combinator-parsing

Theory II

type checking, type inference

Advanced Topics

lazy evaluation, infinite data structures, monads, . . .

Motivation

Idea

Only compute values that are needed for the final result.

Example

In the program

```
let f1 x = x + 1 in

let f2 x = (* something non-terminating *) in

let x    = read_int() in
Lst.hd(f1 x :: f2 x)
```

the value of '`f2 x`' is not needed. Nevertheless, the whole program does not terminate.

Custom Lazy Lists – 1st Iteration

Type

```
type 'a llist = Nil | Cons of ('a * 'a llist)
```

Example

```
Nil                  ([])
Cons(1,Nil)          ([1])
Cons(2,Cons(1,Nil))  ([2;1])
```

Functions

```
let hd = function Nil       -> failwith "empty␣list"
                | Cons(x,_) -> x

let rec from n = Cons(n,from(n+1))
```

Custom Lazy Lists – 1st Iteration (cont'd)

Problem

```
# hd(from 0);;
Stack overflow ...
```

Idea

- block computation of *tail*, until explicitly requested
- use `unit` function (i.e., of type `unit -> ...`)

Custom Lazy Lists – 2nd Iteration

Type

```
type 'a llist = Nil | Cons of ('a * (unit -> 'a llist))
```

Example

```
Nil                                    ([])
Cons(1, fun () -> Nil)                 ([1])
Cons(2, fun () -> Cons(1, fun () -> Nil))  ([2;1])
```

Functions

```
let hd = function Nil        -> failwith "empty␣list"
                | Cons(x,_)  -> x

let tl = function Nil        -> failwith "empty␣list"
                | Cons(_,xs) -> xs ()

let rec from n = Cons(n,fun() -> from(n+1))
```

Custom Lazy Lists – 2nd Iteration (cont'd)

Now

```
# hd(from 0);;
- : int = 0

# hd(tl(from 0));;
- : int = 1
```

But

- strange that *tail* of `llist` is not `llist` itself
- use a mutually recursive type

Custom Lazy Lists – 3rd Iteration (module UnitList)

Type

```
type 'a cell = Nil | Cons of ('a * 'a llist)
and 'a llist = (unit -> 'a cell)
```

Example

```
fun () -> Nil                                       ([])
fun () -> Cons(1,fun () -> Nil)                     ([1])
fun () -> Cons(2,fun () -> Cons(1,fun () -> Nil))   ([2;1])
```

Custom Lazy Lists – 3rd Iteration (module UnitList cont'd)

Functions

```
let hd xs = match xs() with Nil         -> failwith "empty"
                           | Cons(x,_) -> x

let rec from n = fun() -> Cons(n,from(n+1))

let rec to_list n xs = if n < 1 then [] else match xs() with
  | Nil         -> []
  | Cons(x,xs) -> x :: to_list (n-1) xs
```

Example

```
# from 0;;
- : int llist = <fun>

# to_list 10 (from 0);;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
```

Recall

Definition (i-th Fibonacci number F_i)

$$F_i = \begin{cases} 0 & \text{if } i = 0 \\ 1 & \text{if } i = 1 \\ F_{i-1} + F_{i-2} & \text{otherwise} \end{cases}$$

Sequence

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 ...

Idea

Visualization

starting at 0	0	1	1	2	3	5	8	13	21	...
starting at 1	1	1	2	3	5	8	13	21	...	
(+)	1	2	3	5	8	13	21	34	...	

Missing

- function to shift sequence to the left → `tl`
- function to add two sequences → `zip_with (+)`

Implementation (in module UnitList)

```
let tl xs = match xs() with Nil          -> failwith "empty"
                          | Cons(_,xs) -> xs

let rec zip_with f xs ys = fun() -> match (xs(),ys()) with
 | (Cons(x,xs),Cons(y,ys)) -> Cons(f x y,zip_with f xs ys)
 | _                       -> Nil

let rec fibs =
 fun() -> Cons(0,fun() -> Cons(1, zip_with (+) fibs (tl fibs)))
```

Example

```
# to_list 10 fibs
- : int list = [0; 1; 1; 2; 3; 5; 8; 13; 21; 34]
```

Problem

Lazy Enough?

- we defer computation (i.e., call-by-name evaluation) ✓
- we do not use memoization ×

Memoization

- prohibit recomputation of same expressions
- built-in in OCaml's support for lazyness

Lazyness in OCaml

Keyword `lazy`

used to transform arbitrary expression into lazy expression

Example

- `let e = lazy(print_string "test\n")`
- `let f = lazy(let rec f() = print_int 1;f() in f())`

Function `Lazy.force`

used to evaluate lazy expressions

Example

- `Lazy.force e`
- `Lazy.force f`

Lazy Lists Again (module LazyList)

Type

```
type 'a t    = 'a cell Lazy.t
and 'a cell = Nil | Cons of ('a * 'a t)
```

Example

```
lazy Nil                                   ([])
lazy (Cons(1, lazy Nil))                   ([1])
lazy (Cons(2, lazy (Cons(1, lazy Nil))))   ([2;1])
```

The Sieve of Eratosthenes (module LazyList)

Algorithm

start with list of all natural numbers (from 2 on)

1. mark first element *h* as prime
2. remove all multiples of *h*
3. goto Step 1

Functions

```
let fc = Lazy.force

let rec from n = lazy(Cons(n,from(n+1)))

let rec to_list n xs = if n < 1 then [] else match fc xs with
  | Nil         -> []
  | Cons(x,xs) -> x :: to_list (n-1) xs
```

The Sieve of Eratosthenes (module LazyList cont'd)

```
let rec filter p xs = lazy(match fc xs with
 | Nil        -> Nil
 | Cons(x,xs) -> if p x then Cons(x,filter p xs)
                        else fc(filter p xs)
)

let rec sieve xs = lazy(match fc xs with
 | Nil        -> Nil
 | Cons(x,xs) ->
  Cons(x,sieve(filter (fun y -> y mod x <> 0) xs))
)

let primes = sieve(from 2)
```