Functional Programming

WS 2008/09

Christian Sternagel (VO)[1] Friedrich Neurauter (PS)[2]

Computational Logic
Institute of Computer Science
University of Innsbruck

9 January 2009

[1] christian.sternagel@uibk.ac.at
[2] friedrich.neurauter@uibk.ac.at

Type Checking

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

Type Inference

- get the most general type for an expression w.r.t. an environment
- formally: $E \triangleright e : \tau$
- task is split into two parts:
  1. transform given type inference problem into a unification problem
  2. solve the unification problem (result is substitution)

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

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

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)
```

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))
```

Functions

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

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

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

Now

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

But

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

Custom Lazy Lists – 3rd Iteration

Type

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

Functions

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

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

Converting a Lazy List Into a List

Function

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

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
- function to add two sequences

Implementation

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

let rec zip2_with f xs ys = fun() -> match (xs(),ys()) with
 | (Cons(x,xs),Cons(y,ys)) -> Cons(f x y,zip2_with f xs ys)
 | _                       -> Nil

let rec fibs() = Cons(0,fun() -> Cons(1,
  zip2_with (+) fibs (tl fibs)))
```

Problem

Not Lazy Enough

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

Memoization

- prohibit recomputation of equal 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

Type

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

Functions

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

The Sieve of Eratosthenes

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

The Sieve in OCaml

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

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)
```