

# Functional Programming WS 2008/09

### Christian Sternagel (VO)<sup>1</sup> Friedrich Neurauter (PS)<sup>2</sup>

Computational Logic Institute of Computer Science University of Innsbruck

9 January 2009

<sup>1</sup>christian.sternagel@uibk.ac.at <sup>2</sup>friedrich.neurauter@uibk.ac.at

# Type Checking

- prove that some expression really has a given type w.r.t. an environment
- formally:  $E \vdash e : \tau$
- $\blacktriangleright$  use the inference rules of  ${\cal C}$  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 + 1

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

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

the value of 'f2  $\,\mathrm{x}'$  is not needed. Nevertheless, the whole program does not terminate.

# Custom Lazy Lists – 1<sup>st</sup> Iteration

# Custom Lazy Lists – 1<sup>st</sup> 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 – 2<sup>nd</sup> Iteration

## Туре

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

### Functions

```
let hd = function Nil \rightarrow failwith "empty_list"
| Cons(x,_) \rightarrow x
```

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

# Custom Lazy Lists – 2<sup>nd</sup> 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 – $3^{rd}$ Iteration

## Туре

#### 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

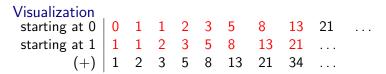
Recall

## Definition (*i*-th Fibonacci number $F_i$ )

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

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

### Idea



### 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

#### Functions

```
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)
```