Lazy Functional Logic Programming and Encapsulated Search

Sebastian Fischer

Not my own Work

Encapsulating Nondeterminism in Functional Logic Computations

Brassel, Hanus, Huch 2004

Computing with Subspaces

Antoy, Brassel 2007

Set Functions for Functional Logic Programming

Antoy, Hanus 2009

The Curry Language

- first-class nondeterminism
- call-by-need (lazy) semantics
- encapsulated search

First-class Nondeterminism

```
coin :: Int
coin = 0 ? 1
```

```
cyi> coin
0
More? yes
1
More? yes
No more results
```

Example: Permutations

```
perm :: [a] -> [a]
perm l =
  if null l then l
  else insert (head l) (perm (tail l))

insert :: a -> [a] -> [a]
insert x xs = (x:xs) ?
  if null xs then fail
  else head xs : insert x (tail xs)
```

Example: Permutations

```
cyi> perm [1,2,3]
[1,2,3]
More? all
[1,3,2]
[2,1,3]
[2,3,1]
[3,1,2]
[3,2,1]
```

First-class Nondeterminism

expressions can have multiple values

interactive environment for examining them

Infinitely Many Results

```
zeros :: Int
zeros = 0 ? zeros

cyi> zeros
0
More? yes
0
More? yes
0
More? no
```

Infinite Values

```
coins :: [Int]
coins = coin : coins

cyi> coins
^C
cyi> head coins
0
More? all
1
```

Laziness

```
isSorted :: [Int] -> Bool
isSorted l =
  if null l || null (tail l) then True
  else head l <= head (tail l)
       && isSorted (tail l)
```

```
ghci> isSorted [0,-1..]
False
```

Lazy Nondeterminism

```
permsort :: [Int] -> [Int]
permsort l =
  let p = perm l in
  if isSorted p then p else fail
```

same value

shared variable

Lazy Nondeterminism

```
cyi> let x = coin in x + x
0
More? yes
2
More? yes
No more results
```

$0 + 0 = 0$

~~$0 + 1 = 1$~~

~~$1 + 0 = 1$~~

$1 + 1 = 2$

Laziness

- infinitely many results
- infinite (intermediate) values
- evaluation order independence

let x = a ? b in e

$\updownarrow$

(let x = a in e) ? (let x = b in e)

Encapsulated Search

primitive operation

values :: a → [a]

idea:

- values coin = [0,1]
- head (values zeros) = 0
- values (a ? b) ≠ values a ? values b

Encapsulated Search

1. Weak Encapsulation
2. Strong Encapsulation
3. Set Functions

Weak Encapsulation

may be nondeterministic

values coin = [0, 1]

but

let x = coin in values x = [0] ? [1]

because x is introduced "outside" of encapsulated expression

Weak Encapsulation

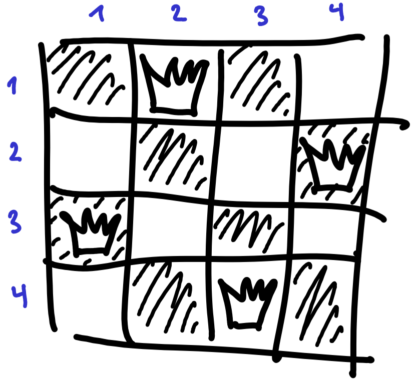
sharing between "inside" and "outside"

```
let x = coin in
  values x ++ (x : values x)
```

↓

[0,0,0] ? [1,1,1]

N - Queens Problem



[3, 1, 4, 2]

N-Queens Problem

```
queens :: Int -> [Int]
queens n =
  let p = perm [1..n] in
  if null (values (capture p)) then p
  else fail

capture :: [Int] -> ()
capture (_++x:xs++y:_) =
 if abs (x-y) == length xs + 1 then () else fail
```

Weak Encapsulation

different results based on
syntactic difference (scope)

not all choices encapsulated

Strong Encapsulation

encapsulates **all** choices

let x = coin in values x

↓

[0, 1]

although x is introduced "outside"
of encapsulated expression

Strong Encapsulation is Reusable

```
hasValue :: a -> Bool
hasValue x = not (null (values x))

firstValue :: a -> a
firstValue x = head (values x)
```

note: x is introduced "outside"

Strong Encapsulation

sharing between "inside" and "outside"

```
let x = coin in values x ++ (x:values x)
```

is **not** [0,1,0,0,1] ? [0,1,1,0,1]

but [0,1,0,0] ? [0,1,1,1]

(in all implementations
of strong encapsulation)

Strong Encapsulation

result depends on evaluation order

let x = coin in values x ++ (x : values x)

if x not yet evaluated

values x = [0, 1]

if x already evaluated to 0 (or 1)

values x = [0] (or [1])

N-Queens

```
queens :: Int -> [Int]
queens n =
  let p = perm [1..n] in
  if p == p && null (values (capture p))
  then p
  else fail
```

↑ force evaluation of p

Strong Encapsulation

- encapsulates all choices
- is reusable
- no evaluation-order independent implementation exists

Set Functions

No primitive `values :: a -> [a]`

Instead: set-valued variant of every defined function

Set Functions

addCoin :: Int → Int
addCoin x = x + coin

generates

$addCoin_{set}$ :: Int → [Int] [1]

[1] conceptually: set, not list

Set Functions

encapsulate choices in body,
but not in arguments

$addCoin_{set}$ (10 ? 20)

↓

[10, 11] ? [20, 21]

choice between 0 and 1

choice between 10 and 20

N-Queens

```
queens :: Int -> [Int]
queens n =
  let p = perm [1..n] in
  if null (capture_set p) then p
  else fail
```

Set Functions

similar to weak encapsulation

but separation of choices
based on argument-body
distinction
rather than on scoping

Lazy Functional Logic Programming

- first-class nondeterminism
- evaluation-order independent call-by-need semantics
- interactive environment for examining results
- encapsulated search

Weak Encapsulation

depends on scoping

not reusable

Strong Encapsulation

reusable

depends on evaluation order
(at least as implemented for Curry)

Set Functions

evaluation-order independent

no (reusable) strong encapsulation

Delimited Continuations ?

fail = []

x ? y = shift k . k x ++ k y

reset : weak or strong encapsulation ?

evaluation order independence ?

ありがとう