Follow the programming?

- Open the VM
- ./login
- cd rfun/
- git pull
- make

I made the repository public, just for you.

Faculty of Science

Janus meets Backstroke

Michael Kirkedal Thomsen

Department of Computer Science, University of Copenhagen

Partly joint work with Holger Bock Axelsen, Tetsuo Yokoyama, Robert GlÃ¼ck

COST Training School Torun

Nicolaus Copernicus University, Aug 28-31 2017

Overview

- Development of reversible programs
  - Embedding a function
- Introduction to RFun
- Development in RFun
  - Arithmetic
  - List functions
  - Equality / duplication

Tutorial can be found at:
http://topps.diku.dk/pirc/rfundocs/

Embeddings

Thinking reversibly

$2 + 3 = 5$

Thinking reversibly

$$2 + 3 = 5$$

$$5 = A + B$$

Thinking reversibly

$2 + 3 = 5$

$5 = A + B$

Reversible embeddings:

$+(2, 3) = ([(2, 3), (3, 2), (4, 1), (5, 0)], 5)$

Thinking reversibly

$2 + 3 = 5$

$5 = A + B$

Reversible embeddings:

$+(2, 3) = ([(2, 3), (3, 2), (4, 1), (5, 0)], 5)$

$+(2, 3) = ((2, 3), 5)$

Thinking reversibly

$2 + 3 = 5$

$5 = A + B$

Reversible embeddings:

$+(2,3) = ([(2,3),(3,2),(4,1),(5,0)],5)$

$+(2,3) = ((2,3),5)$

Embedded addition

$+(x,y) = (x, y + x)$

Reversible embeddings

- How to get a reversible program from a known irreversible program?
- Can we do it automatically?
  - I.e. using program transformation

Embeddings

- Landauer embedding
- Bennett embedding
- (Incremental check-pointing)

Landauer embedding

$$+(x, y) = ([...trace...], x + y)$$

$$+(2, 3) = ([(2, 3), (3, 2), (4, 1), (5, 0)], 5)$$

- Also called trace embedding
  - First (indirectly) prosed by Landauer
- Store a needed amount of information every time you make a irreversible choice
- Will have size equivalent to run-time

Bennett embedding

$$+(x, y) = ((x, y), x + y)$$

$$+(2, 3) = ((2, 3), 5)$$

- Input-output embedding
- First proposed by Bennett
- Can be implemented by Bennett's compute-copy-uncompute method
- Size is limited by input/output
  - but computational space is at the size of Landauer embedding
- Cannot be cascaded

Reversible implementation

$$+(x, y) = (x, x + y)$$

If function injective

- Know reversible implementation exist
  - McCarthy's generate-and-test approach
- Can still be hard to find

If function non-injective

- We have to redefine out problem

Considering your semantics

$$+(x, y) = ([...trace...], x + y)$$

$$+(x, y) = ((x, y), x + y)$$

$$+(x, y) = (x, x + y)$$

Garbage

- Semantically undesired values

Ancillae

- Values that are *guaranteed* unchanged over a computation

RFun

Fibonacci

```
procedure fib(int x1, int x2, int n)
    if n = 0 then
        x1 += 1
        x2 += 1
    else
        n -= 1
        call fib(x1, x2, n)
        x1 += x2
        x1 <=> x2
    fi x1 = x2
```

Fibonacci

```
procedure fib(int x1, int x2, int n)
    if n = 0 then
        x1 += 1
        x2 += 1
    else
        n -= 1
        call fib(x1, x2, n)
        x1 += x2
        x1 <=> x2
    fi x1 = x2
```

Comparison from Janus

```
fib :: Nat <-> (Nat, Nat)
fib Z     = ((S Z),(S Z))
fib (S m) =
  let (x,y) = fib m
      y' = plus x y
  in (y',x)
```

RFun

- A history-free functional reversible language
- Implements (often) injective partial functions
  - I.e. we are usually not implementing in bijections

Background

- First formalised in 2012 by Tetsuo Yokoyama, Robert GlÃ¼ck, and Holger Bock Axelsen [1]
  - Untyped, first-order language based on constructor terms
- Implemented and explored by Michael Kirkedal Thomsen and Holger Bock Axelsen [2]
  - Extension to (somewhat) second-order language with syntactical support for tuples, lists, and natural numbers
- New version designed for this training school by Michael Kirkedal Thomsen
  - Updated syntax and extended with type system

Important concepts

- Linearity
- Ancillae
- First-match policy

Important concepts - Linear typing

- Stems from linear logic
- We **must** use a resource exactly once

Examples of usage

- Handling of memory resources
- Seen in many modern language
    - most recently Rust

Linearity - explained

```
fib :: Nat <-> (Nat, Nat)
fib Z     = ((S Z),(S Z))
fib (S m) =
  let (x,y) = fib m
      y' = plus x y
  in (y',x)
```

- Variable `m`
  - introduced on the left-hand-side
  - used exactly in the recursive call to `fib`.
- Similarly for `y` and `y'`
  - also consider the return of `y'` as a usage.

Important concepts - Ancillae type

- Variables for which we can *guarantee* that the value is unchanged over a function call.
- The *guarantee* is important
  - using a conservative approach.

Also seen in

- Reversible logic
- Restore model

Ancillae - explained

```
fib :: Nat <-> (Nat, Nat)
fib Z     = ((S Z),(S Z))
fib (S m) =
  let (x,y) = fib m
      y'  = plus x y
  in (y',x)
```

- Variable `x`
  - introduced by recursive call to `fib`
  - used by the `plus` function
  - returned by the `fib` function.
- Here `plus` is using `x` as an ancillae

Important concepts - First-match policy

- important to guarantee injectivity of the functions
- conceptually, function must not return value that can be the result of any previous branches
  - knowing nothing about the possible content of variables.

Two sides to the coin

- Pattern matching for clauses in inverse interpretation
- Alternative to Janus assertions

First-match policy - detailed

$$
\begin{array}{l}
\textbf{case } l \textbf{ of} \\
\quad \begin{array}{lcl}
l_1 & \rightarrow & \cdots \textbf{ in } l'_1 \\
 & \vdots & \\
l_i & \rightarrow & \cdots \textbf{ in } l'_i \\
 & \vdots & \\
l_n & \rightarrow & \cdots \textbf{ in } l'_n
\end{array}
\end{array}
$$

- The value v of l is matched against the left-hand side of each branch $(l_1, l_2, \ldots)$ until the first successful match l_i.
- The right-hand side of the i-th branch is then evaluated in σ_i and a value v' is returned by l'_i.
- Now, for symmetry, v' must not match any of the preceding $l'_1, \ldots, l'_{i-1}$
- Otherwise, the case-expression is undefined.

First-match policy - explained

```
procedure fib(int x1, int x2, int n)
    if n = 0 then
        x1 += 1
        x2 += 1
    else
        n -= 1
        call fib(x1, x2, n)
        x1 += x2
        x1 <=> x2
    fi x1 = x2
```

```
fib :: Nat <-> (Nat, Nat)
fib Z      = ((S Z),(S Z))
fib (S m) =
  let (x,y) = fib m
      y' = plus x y
  in (y',x)
```

- Base-clause matches the non-recursive branch

First-match policy - explained

```
fib :: Nat <-> (Nat, Nat)
fib Z     = ((S Z),(S Z))
fib (S m) =
  let (x,y) = fib m
      y' = plus x y
  in (y',x)
```

- FMP can often be check statically
  - based on the type definitions
  - This is not yet implemented in RFun
- FMP is in general impossible to check statically.

E.g. `fib` **cannot** be statically checked

Implementation in RFun

Natual number arithmetic

Define Peano numbers

```
data Nat = Z | (S Nat)
```

- Read: a natural number (`Nat`) is either zero (`Z`) or the successor of a natural number (`(S Nat)`).

Type for simple incremental function

```
inc :: Nat <-> Nat
```

- Read: increment (`inc`) is a (reversible) function that transforms a `Nat` to a `Nat`

Natual number arithmetic

```
data Nat = Z | (S Nat)
inc :: Nat <-> Nat
```

Implementation if `inc`

Natual number arithmetic

```
data Nat = Z | (S Nat)
inc :: Nat <-> Nat
```

Implementation if `inc`

```
inc n = (S n)
```

- Read: `inc` given a natural number `n` returns the successor of `n`.

Natual number arithmetic

```
data Nat = Z | (S Nat)
inc :: Nat <-> Nat
inc n = (S n)
```

Decremental can be implemented as

```
dec :: Nat <-> Nat
```

Natual number arithmetic

```
data Nat = Z | (S Nat)
inc :: Nat <-> Nat
inc n = (S n)
```

Decremental can be implemented as

```
dec :: Nat <-> Nat
dec (S n) = n
```

Natual number arithmetic

```
data Nat = Z | (S Nat)
inc :: Nat <-> Nat
inc n = (S n)
```

Decremental can be implemented as

```
dec :: Nat <-> Nat
dec (S n) = n
```

or even better

```
dec :: Nat <-> Nat
dec n = inc! n
```

- `!` at the end of the function specifies reverse execution.

Reversible Addition

$$+(x, y) = (x, y + x)$$

Type for addition?

Reversible Addition

$$+(x, y) = (x, y + x)$$

Type for addition

```
plus' :: (Nat, Nat) <-> (Nat, Nat)
```

Reversible Addition

$$+(x, y) = (x, y + x)$$

Type for addition

```
plus' :: (Nat, Nat) <-> (Nat, Nat)
```

Considering that x is an ancilla value

```
plus :: Nat -> Nat <-> Nat
```

- Read: `plus` is a function that given a natural number (`Nat`), will transform one natural number (`Nat`) to another natural number (`Nat`).

Reversible Addition

Implementation

```
plus :: Nat -> Nat <-> Nat
plus Z     x = x
plus (S y) x =
  let x' = plus y x
  in  (S x')
```

Reversible Addition

Implementation

```
plus :: Nat -> Nat <-> Nat
plus Z     x = x
plus (S y) x =
  let x' = plus y x
  in  (S x')
```

- Why is the first argument guaranteed to be unchanged?
- Why is the second argument linear?
- Why is the first-match policy upheld?

Reversible Addition

```
plus :: Nat -> Nat <-> Nat
plus Z     x = x
plus (S y) x =
  let x' = plus y x
  in  (S x')
```

- Why is the first argument guaranteed to be unchanged?
  - `y` is only used as an ancilla in the recursive call
- Why is the second argument linear?
  - Both `x` and `x'` is first introduced then used
- Why is the first-match policy upheld?
  - First (ancilla) argument of two clauses are disjoint.

Addition - FMP explained

```
plus :: Nat -> Nat <-> Nat
plus Z     x = x
plus (S y) x =
  let x' = plus y x
  in  (S x')
```

We transform `plus` into

```
plusP :: (Nat, Nat) <-> (Nat, Nat)
plusP (Z,    x) = (Z, x)
plusP (S(y), x) =
  let (y, x') = plusP (y, x)
  in  (S(y), S(x'))
```

- Here is it clear that the first element of the branch tuples are disjoint, making the entire tuple disjoint.

Addition transformation explained

```
plusP :: (Nat, Nat) <-> (Nat, Nat)
plusP (Z,    x) = (Z, x)
plusP (S(y), x) =
  let (y, x') = plusP (y, x)
  in  (S(y), S(x'))
```

How do we generate `plusP`?

- wrap our input into a tuple,
- add the ancillae arguments to all output leaves,
- wrap all function calls into tuples,
- add the ancillae inputs to function calls to the output.

Transformation of ancilla functions

With the above method, we can always transform

```
f :: a -> b <-> c
```

into

```
f :: (a, b) <-> (a, c)
```

- This does not (in general) work in the opposite direction.

List functions

- List is a predefined type in RFun
- Use standard notation
    - `[` and `]` for list specification
    - `(l : ls)` for list construction

Examples

- List of 5 elements and empty list
    - `[1,2,3,4,5]` and `[]`
- A list construction
    - `(1:2:[3,4,5])`

The lenght of a list

The type of this is

```
length :: [a] <-> Nat
```

!!!!

The lenght of a list

The type of this is

```
length_wrong :: [a] <-> Nat
```

- Length should be treated as a property of a list
  - I.e. given a list we need to extract the information

```
length :: [a] -> () <-> Nat
```

- Read: `length` is a function that given a list, transforms nothing to a natural number.
  - Empty tuple does not contain any information
- I.e. we are making a Bennett embedding of the normal length function.

Implementation of length

We can then implement length as the standard

```
length :: [a] -> () <-> Nat
length []       () = Z
length (x : xs) () =
  let  n = length xs ()
  in   (S n)
```

- Why is the input list guaranteed to be ancillae?
- How is linear typing upheld?
- How is FMP upheld?

Implementation of length

We can then implement length as the standard

```
length :: [a] -> () <-> Nat
length []       () = Z
length (x : xs) () =
  let  n = length xs ()
  in   (S n)
```

- Why is the input list guaranteed to be ancillae?
  - `xs` is only used for ancilla to `length`
- How is linear typing upheld?
  - The introduced variable `n` is used again
- How is FMP upheld?
  - `Z` is disjoint from `(S Nat)`

Mapping functon over list

```
map :: (a <-> b) -> [a] <-> [b]
```

- Read: given a function that transforms `a`'s to `b`'s, `map` will transform a list of `a`'s to a list of `b`'s.
  - This is exactly how we consider the normal map-function.

Mapping functon over list

Implementation

```
map :: (a <-> b) -> [a] <-> [b]
map fun      []    = []
map fun (x : xs) =
  let x'   = fun x
      xs' = map fun xs
  in  (x'  : xs')
```

- Ancilla of the mapped function?
- Linearity of the lists?
- Is FMP upheld?

List reversal

```
reverse :: [a] <-> [a]
```

Two different approaches

- Appending a first element of a list to the end of a reversed list
    - Squared to the length of the list run-time
- Using an accumulator
    - Linear to the length of the list run-time
    - Helper function moves elements from one list to the other

We will go for the fast version.

List reversal

Function from Haskell

```
reverse_haskell l =  rev l []
  where
    rev []     a = a
    rev (x:xs) a = rev xs (x:a)
```

List reversal

Function from Haskell

```
reverse_haskell l =  rev l []
  where
    rev []     a = a
    rev (x:xs) a = rev xs (x:a)
```

Reversible implementation of `rev`

```
rev :: ([a], [a]) <-> ([a], [a])
rev (  []  , l) = ([], l)
rev ((x:xs), l) = move (xs, (x:l))
```

- Is linearity guarenteed?
- Is FMP upheld?

Moving elements

Instead lets make a function that moves some elements

```
move :: Nat -> ([a], [a]) <-> ([a], [a])
move  Z     (    x   , l) = (x, l)
move (S s) ((x:xs), l) = move s (xs,(x:l))
```

- Is Ancilla types guaranteed?
- Is linearity guaranteed?
- Is FMP upheld?

List reversal

Final reversal of list

```
reverse :: [a] <-> [a]
reverse xs =
  let xs_s = length xs ()
      ([], ys) = move xs_s (xs, [])
      () = length! ys xs_s
  in  ys
```

- How is Ancilla types guaranteed?
- Is linearity guaranteed?
- Is FMP upheld?
- Is this linear run-time to the size of the list?

Equality and Duplication

Equality (and duplication) have a special place in FRun.

- Predefined type

```
data EQ = Eq | Neq a
```

- This is equivalent to the definition of Maybe monad from Haskell
  - We will not use it as a monad...

Predefine function of type

```
eq :: a -> a <-> EQ
```

Equality and Duplication

```
data EQ = Eq | Neq a
eq :: a -> a <-> EQ
```

Given `eq x y`,

- first argument (`x`) is ancillae
- second argument (`y`), will be transformed into a `EQ` type, where the result is
  - `Eq` if `x` is equal to `y`
  - `Neq y` if `x` is different from `y`.

Note, equality can remove one copy of the two values.

- No implementation of `eq` is possible in RFun.

Equality and Duplication

```
data EQ = Eq | Neq a
eq :: a -> a <-> EQ
```

Based on `eq` we can then make a duplication function by inverse execution

```
dup :: a -> () <-> a
dup x () =  eq! x Eq
```

Run-length encoding

```
pack :: [a] <-> [(a, Nat)]
pack [] = []
pack (c1 : r) =
  case (pack r) of
    [] -> [(c1, 1)]
    ((c2, n) : t) ->
      case (eq c1 c2) of
        (Neq c2p) -> ((c1, 1) : (c2p, n) : t)
        (Eq) -> ((c1, (S n)) : t)
```

- Notice usage of `eq`

Example of Running it

```
pack [1,1,3,2,2,2] = [(1,2),(3,1),(2,3)]
```

- Why is FMP upheld? Can we statically check it?

Summing of RFun

RFun

- A history-free functional reversible language
- Implements (often) injective partial functions
  - I.e. we are usually not implementing in bijections
- Looks like a functional language
- First steps toward higher-order language
- Type system with linear and ancilla types
- Support for tuples and lists

Future work and extensions

- Int type
- Guards
- Static check of FMP
- Support for kinds; especially with usage for `eq`
- Partial applications and higher-order functions

Arithmetic Exercises

- Implement a `minus` function
- Implement a function that multiplies by 2
- Implement a multiplication function **(hard)**
  - What embedding makes sense?
- Implement a function `even`, that checks if a natural number is even
  - What embedding of `even` makes sense?

Arithmetic List

- Implement a function that checks if all elements of a list is even
- Implement `splitAt` that splits a list in two after a given lenght
    - What is a good embedding?
    - You can find the Haskell definition here http://hackage.haskell.org/package/base-4.10.0.0/docs/Prelude.html#v:splitAt
- Implement an `append` function
    - Again, the embedding is not clear?

Arithmetic List - continued

- - Implement `scanl` and `scanr` **(hard)**
  - You can find the Haskell definition here http://hackage.haskell.org/package/base-4.10.0.0/docs/Prelude.html#v:scanl
- Implement `foldl` and `foldr` **(even hard)**
  - What is a resonable embedding?
  - How does this differ from the scan's
- Continue with `interleave`, `intercalate`, `permutations`, `group`, `subsequences`, `transpose`, ...

References

[1] T. Yokoyama and H. B. Axelsen and R. Gluck, Towards a reversible functional language, Reversible Computation, RC '11, 7165 14--29 (2012)

[2] M. K. Thomsen and H. B. Axelsen, Interpretation and Programming of the Reversible Functional Language, Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages, 8:1--8:13 (2016)

[3] M. K. Thomsen, RFun tutorial, http://topps.diku.dk/pirc/rfundocs/

Thank you!