Functional Languages 101

What's All the Fuss About?

Rebecca Parsons
ThoughtWorks

Agenda

- What makes a language functional?
- An larger example
- The case for renewed interest in functional languages
- Other neat and nifty things to do with functional languages
- Resources for further study

Some Example Languages

- Scheme, Lisp (and of course Lambda Calculus) – the originals
- ML, Ocaml, etc   - here comes typing!
- Haskell – a lazy language not for the lazy
- Erlang – message passing
- Scala, Clojure, F# - the new(er) kids on the block

Essentials of Scheme

- (define name expr)
- (func arg…) | ident | symbol | (lambda (x) e)
- +/-/*, cond, eq? #t
- Lists, car/cdr/cons, null?

Two Examples

```
(define length
  (lambda (ll)
    (cond
      ((null? ll) 0)
      (#t (add1
            (length (cdr ll)))))))
```

```
(define squares
  (lambda (li)
    (cond
      ((null? li) ())
      (#t (cons
            (* (car li) (car li))
              (squares (cdr
li))))))))
```

Characteristics

- No side effects (for some definition of No)
  - Values looked up in an environment
- Functions as first class citizens
- Composition of functions, not statements
- Type inference

A function accepts some number of arguments, each of an appropriate type, and returns a result of the appropriate type

A computation is referentially transparent if invoking it on the same input values always returns the same result

Observation

Computation involving mutable state can not be referentially transparent

```
int state = 10;

int foo (int bar) {

  state = state + bar;

  return (state);
}

foo (10) => 20
foo (10) => 30
```

we can’t reason about the value of foo(10) anymore. And oh yeah, btw, my call to foo can mess up your call if we share state.

Side Effects and Mutable State

- Both complicate reasoning about program behavior.
- However, that doesn't mean we can do without side effects
  - Persistence
  - Dispensing cash
  - Requesting input
  - Displaying a page

Functions Can Be…

- Passed as arguments
- Created at run time and then used
- Returned as values
- Assigned to variables

- Yup – they're just like any other data type!

In fact, in pure Lambda Calculus, integers are functions too, but I digress

Even constructs like *if* can be a function too, taking three arguments – conditional expression, true expression and false expression, with suitable thunking or laziness.

Functions in the Functional World

- Higher order functions
- Currying
- Closures
- Iteration, recursion and tail recursion
- Function maker

A more complicated example

```
(define map (lambda (func lst)
   (cond ((null? lst) ())
            (#t (cons (func (car lst))
                        (map func (cdr lst)))))))

(map add1 ‘(0 1 2 3))
(1 2 3 4)
(map length ‘((1 2 3) (a b) (ola amanda john
aino)))
(3 2 4)
```

Curried Functions

- No, not Indian cuisine
- Partial application of a function
  - Let's think in types for a moment.
- Func-a: (AxB) -> C
- Func-b: A -> (B -> C)
  - Such that (Func-a x y) = ((func-b x) y)

- Let's look at map a bit differently

Map again

```
(define map (lambda (func lst)
   (cond ((null? lst) ())
          (#t (cons (func (car lst))
                    (map func (cdr lst)))))))
```

A Curried Version

```
(define map2
  (lambda (func)
    (lambda (lst)
        (cond ((null? lst) ())
              (#t (cons (func (car lst))
                        ((map2 func) (cdr lst)))))))

(define list-count (map2 length)
(list-count '((1 2 3) (a b) (amanda john ola
aino)))

(3 2 4)
```

Guess what? I just made a closure!

See how easy that was.

So what is a closure?

- A functional data object (which I can pass around) …
- … with a local environment that comes from its lexical scope (in Sheme)
- The result of (map2 length) is a closure …
- … func is bound in that closure to length
- A closure is a function with a local environment (mapping identifiers to values)

And I care why?

- Closures allow the easy creation of functions during run-time
- Closures, and more generally higher order functions, is an important part of the expressiveness of functional languages

Writing a functional program

- Basic unit of design is a function
- Recursion is fundamental (base case(s) and recursive step(s))
- Rely on compiler to transform and optimize tail recursion
- Think of the problem as successive transformations of data items by functions
- Easiest to think in terms of recursive and compound data structures

Function Maker

- This style of programming results in recurring patterns in code
- Remember squares and length?

Two Examples (repeated)

```
(define length
  (lambda (ll)
    (cond
      ((null? ll) 0)
      (#t (add1
              (length (cdr ll)))))))
```

```
(define squares
  (lambda (li)
    (cond
      ((null? li) ())
      (#t (cons
              (* (car li) (car li))
                (squares (cdr
li)))))))
```

Squares and Length

- Base case of the recursive call is null?
- Recursion step based on some function of the car of the list and recursion on the cdr.
  - I did cheat a bit here.
- Can we generalize?

```
(define list-function-mker
  (lambda (base rec-op)
    (lambda (ll)
      (cond
        ((null? ll) base)
        (#t (rec-op (car ll) ((list-function-mker base rec-op) (cdr ll))))))))

(define sq2 (list-function-mker ()
                                (lambda (num results)
                                  (cons (* num num) results))))

(define len2 (list-function-mker 0
                                 (lambda (head result)
                                   (add1 result))))
```

- Patterns are expressible in such function-makers
- They can be instantiated with functions in the various slots
- Significantly reduces code duplication
- Still very easy to unit test
- Somewhat more difficult to read, until you get used to it

A Bigger Example

- Parser combinators
  - Pairing of recognizer and semantic model construction on recognition
- Use matchers for terminal symbols
- Combine these using combinators for the grammar operations
  - | is alternative
  - Sequence
  - Various closures (*, +, n)

How to Construct Sequences

- The sequence combinator
  - .. accepts a list of combinators and a model function
  - .. returns a closure binding that combinator list and accepting a token buffer
  - If all combinators match as the tokens are consumed, then the model function is applied to the list of models from the combinators.
  - The final token buffer is returned, with all the matched tokens consumed.

So, why now?

- Increasing need for concurrent programming (multi-core)
  – To get speed increases, need to exploit cores
  – But parallel programming in imperative languages is hard (deadlocks, race conditions, etc)
  – Functional programs don’t share state so they can’t trample on anyone else’s state
- Please remember though, you still have to design the program to run concurrently
  – It doesn’t come for free

Other things to explore

- Continuations:
  - A continuation is a function that represents the rest of the computation
  - (* (+ 1 2) 3) can be thought of as a redux (+ 1 2) and the continuation (* [] 3)
- And what are they good for?
  - Can be used for exception management
  - Can be used for workflow processing
  - Optimization (0 in list multiplication)

Lazy Languages

- Strict computation proceeds by evaluating all arguments to a function and then invoking the function on the resulting values
- Lazy computation proceeds by not evaluating argument values unless and until they are actually needed (occur in a strict position
- Examples of strict positions: first position of function application, conditional expression in a control structure, anything going external like a print statement

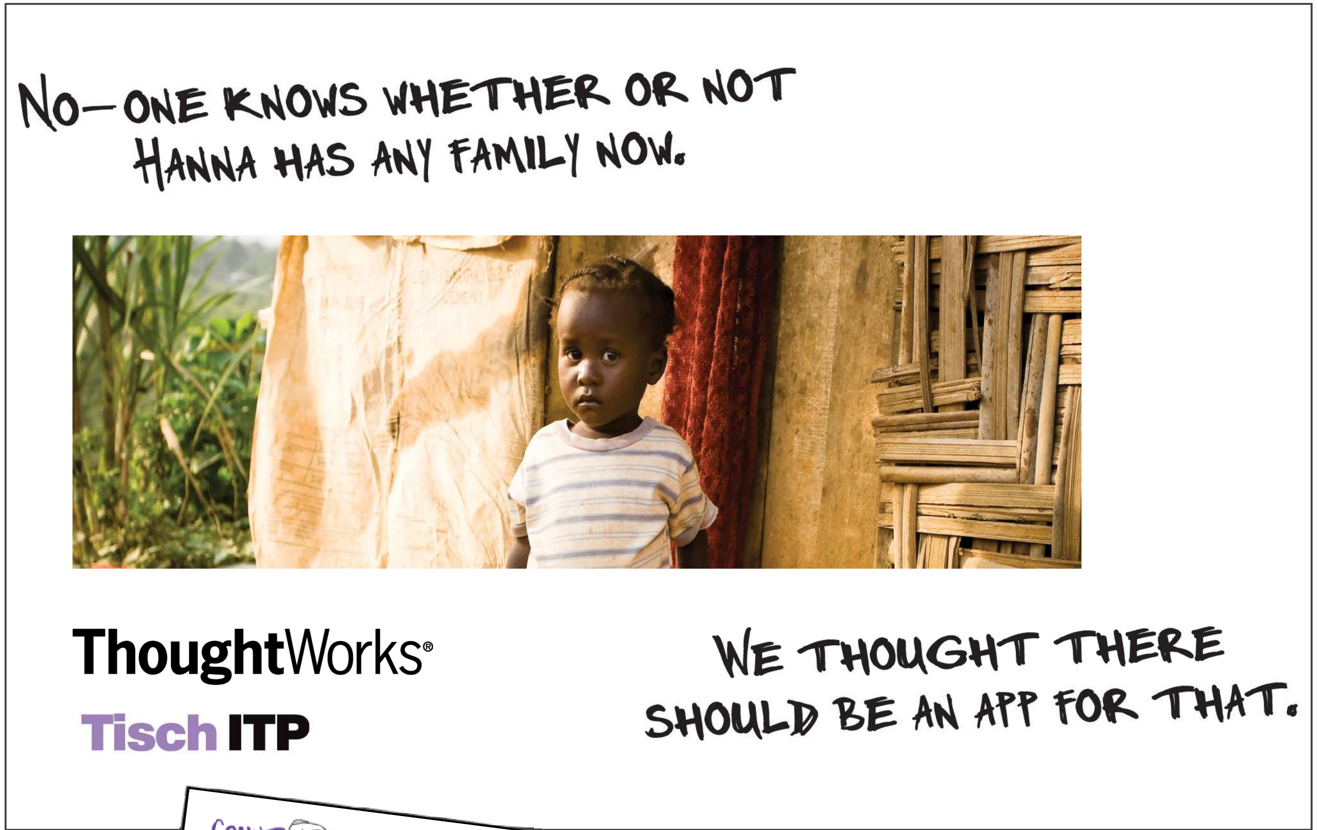
Type Systems

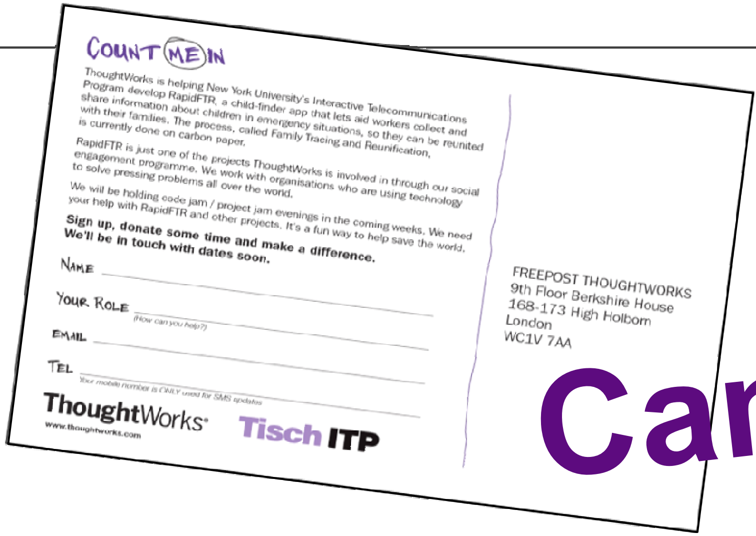
- Static versus dynamic typing
  - Yes, the war is still raging
- The joys of type inference
  - Why should you have to specify the type of everything?
  - Type systems can be helpful (I guess)

Resources

- Dr. Scheme and the PLT web site www.pltscheme.org
- Structure and Interpretation of Computer Programs www.mitpress.mit.edu/sicp
- Haskell www.haskell.org and Programming Haskell
- Caml and Ocaml http://caml.inria.fr/
- Erlang www.erlang.org
- F# (Microsoft, F# in Action)

QUESTIONS???





Count ME in

ThoughtWorks is helping New York University's Interactive Telecommunications Program develop RapidFTR, a child-finder app that lets aid workers collect and share information about children in emergency situations, so they can be reunited with their families. The process, called Family Tracing and Reunification, is currently done on carbon paper.

RapidFTR is just one of the projects ThoughtWorks is involved in through our social engagement programme. We work with organisations who are using technology to solve pressing problems all over the world.

We will be holding code jam / project jam evenings in the coming weeks. We need your help with RapidFTR and other projects. It's a fun way to help save the world.

Sign up, donate some time and make a difference.
We'll be in touch with dates soon.

Name ______

Your Role ______
(How can you help?)

Email ______

Tel ______
Your mobile number is ONLY used for SMS updates

ThoughtWorks
www.thoughtworks.com

Tisch ITP

FREEPOST THOUGHTWORKS
9th Floor Berkshire House
168-173 High Holborn
London
WC1V 7AA

Can we count you in ?