Learning how to Compute
...without a Computer!

Andrea Falconi
andrea.falconi@gmail.com

Dynamic Technology Holdings

May 2009

Learning how to Compute ... without a Computer!

Goals

- To introduce fundamental concepts of "computing with functions".
- To start developing a taste for functional abstraction.

Topics

- Functions and types.
- Function composition.
- Induction and recursion.
- Recursive definitions of functions.
- Functional evaluation model.
- Lazy evaluation.
- Importance of proof.

Lost in Change

Our Task

Write a program that, given a list of persons, tells you whether or not there's at least one person who likes functional programming and has green eyes.

We'll develop it following different approaches:

- Imperative: tell the computer what to do by sequencing commands that change the state of the program data.
- Purely Functional: evaluate mathematical functions; ban state changes.

Lost in Change

Main Course

```
class Color
    ...
    bool isGreen()

class Person
    ...
    bool likesFP()
    Color eyeColor()

class Finder
    List<Person> whoLikesFP(List<Person> ppl)
        for (k = 0; k < ppl.size; ++k)
            Person p = ppl.get(k);
            if (!p.likesFP) ppl.remove(k);
        return ppl;

    bool someoneGreenEyed(List<Person> ppl)
        for (k = 0; k < ppl.size; ++k)
            Person p = ppl.get(k);
            if (!p.eyeColor.isGreen) return true;
        return false;
```

Lost in Change

Side Dishes

```
void main()
    Joe   = new Person(likesFP = true,  eyeColor = Black),
    Tom   = new Person(likesFP = false, eyeColor = Green),
    Sally = new Person(likesFP = true,  eyeColor = Green);

    List<Person> ppl = [ Joe, Tom, Sally ];

    List<Person> fpFolks = Finder.whoLikesFP(ppl);

    // Now:     ppl = [ Joe, Sally ]     fpFolks = [ Joe ]

    bool answer = Finder.someoneGreenEyed(fpFolks);

    print(answer);        // ⇝ false
```

Lost in Change

Even a conceptual simple task can become fairly involved to program.

- Have to think about how state changes in one place are going to affect other pieces of code.
- Have to keep track of all of variables, objects, etc. operated on by the code at each computation step.
- Have to consider all possible combinations of state changes, but number usually fairly large.
- Invariably, things crop up that we had not seen or thought about initially.

To Be or not to Be-come

- How to eliminate (or at least contain) side-effects?
- If state changes cause side-effects, then ban state changes.
- Nothing outside of a code unit can possibly affect it nor can its execution cause side-effects.
- Move away from seeing variables as “buckets” to store values.

To Be or not to Be-come

Equal is Equal

In `for` loop above, does $k = 0$ really means *equal*?

- '=' means "assign" and k is a "bucket" to store a value.
- State of the bucket (i.e. contained value) may change as statements are executed.
- To avoid state changes we have to get rid of assignment.

Going to use *equations* instead of assignments.

- '=' means: expression on lhs is *identical* to the one on rhs.
- E.g. $k = 2$ means: k is another name for 2, in enclosing eval context.
- Can't give another "meaning" to k, such as $k = 11$.
- Likewise, if ppl = [Joe, Tom, Sally], ppl will always refer to that list and the list may not be changed — e.g. can't remove Tom.

To Be or not to Be-come

Equals by Equals

So we have equations by how do we use them?

- Trivial program: multiply 5 by 2 and add 1.
- Corresponding equation: $result = 5 \times 2 + 1$.
- Parameterized: $result = 5 \times x + 1$.

What mechanism should the program use to compute the result? E.g. if we start from:

$$\begin{aligned} x &= 2 \\ result &= 5 \times x + 1 \end{aligned}$$

how would the program arrive at 11 — i.e. the right answer?

To Be or not to Be-come

Equals by Equals

Computation is a plain matter of *rewriting expressions* by replacing equals with equals until the expressions are reduced to the desired result.

$$
\begin{aligned}
\mathit{result} &= 5 \times x + 1 \\
&= 5 \times 2 + 1 && \text{by equation } x = 2 \\
&= 10 + 1 && \text{by arithmetic} \\
&= 11 && \text{by arithmetic.}
\end{aligned}
$$

A program is nothing more than an expression and running the program consists of evaluating that expression on the provided input.

To Be or not to Be-come

Variables Vary Not

Meaning of a variable in an expression:

- Placeholder for a "typical" value of a certain kind.
- May not be "assigned" different values overtime, but can be bound to a certain value for the sake of evaluating an expression.
- As seen above: x is *substituted* with 2.

Equational Reasoning is then possible:

- Equals can be replaced by equals.
- Can "safely" evaluate an expression and replace it with its value to yield an equivalent program.
- No side-effects in rewriting the expression.

Crunching Values

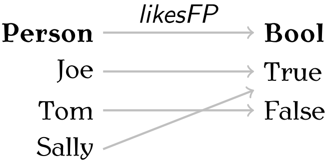
How to re-implement the program in functional land:

- Specify "recipes" to transform an input value into an output value.
- Each recipe defined in terms of equations whose expressions manipulate input to produce output.
- No state changes are involved.
- Specify that to a given input value *corresponds* a specific output value.
- Recipes can be chained together: program's input transformed in subsequent steps, until output is produced.
- Computation proceeds by using those recipes' equations to rewrite initial program expression.

Crunching Values

Basic Recipes

A *rule* to *associate* a Bool to *each* Person, depending on whether they like functional programming.



likesFP Joe = True
likesFP Tom = False
likesFP Sally = True

likesFP Tom || *likesFP* Sally

= False || *likesFP* Sally	by 2^{nd} likesFP equation
= *likesFP* Sally	by || definition
= True	by 3^{rd} likesFP equation.

Crunching Values

Basic Recipes

A *rule* to *associate* an eye Color to *each* Person.

$$
\begin{array}{rcl}
\textbf{Person} & \xrightarrow{\textit{eyeColor}} & \textbf{Color} \\
\text{Joe} & \longrightarrow & \text{Black} \\
\text{Tom} & \longrightarrow & \text{Green} \\
\text{Sally} & \nearrow & \text{Orange} \\
 & & \ldots
\end{array}
$$

```
eyeColor Joe   = Black
eyeColor Tom   = Green
eyeColor Sally = Green
```

And one to tell us whether or not a color is Green.

$$
\begin{array}{rcl}
\textbf{Color} & \xrightarrow{\textit{isGreen}} & \textbf{Bool} \\
\text{Green} & \longrightarrow & \text{True} \\
\text{Black} & \longrightarrow & \text{False} \\
\text{Orange} & \nearrow & \\
\ldots & \nearrow &
\end{array}
$$

```
isGreen Green = True
isGreen _     = False
```

Crunching Values

The Function Club

We've done the same thing over and over again to specify a rule:

- given an input value, decide what is the corresponding output value;
- specified, by means of equations, an input/output transformation that to *each* given input, associated *one and only one* definite output.

Function is the name of the game!

Crunching Values

The Function Club

All functions defined above use enumeration procedure, but not always convenient/possible.
Example:

$$\begin{array}{llcl} \mathit{isOdd} & 0 & = & \mathrm{False} \\ \mathit{isOdd} & 1 & = & \mathrm{True} \\ \mathit{isOdd} & 2 & = & \mathrm{False} \\ \mathit{isOdd} & 3 & = & \mathrm{True} \\ & & \ldots & \end{array}$$

Not going to have an easy time if we go down this road!

Crunching Values

The Function Club

Way out: write down how the function operates on a typical input value.

$$\textbf{isOdd}\ x \ = \ (x\ \text{`}\textbf{mod}\text{`}\ 2) == 1$$

Conceptually *isOdd* same as functions defined previously, still a rule to associate input to output. Just a smarter definition.

When evaluating, substitute placeholder argument with actual input value.

$$\begin{array}{lll} \multicolumn{3}{l}{\mathit{isOdd}\ 4} \\ = & (4\ \text{`}\mathit{mod}\text{`}\ 2) == 1 & \text{by substitution} \\ = & 0 == 1 & \text{by } \mathit{mod} \text{ definition} \\ = & \text{False} & \text{by } == \text{ definition.} \end{array}$$

Crunching Values

Who's Your Type

A function transforms an input value into an output value, but where do these values come from?

- Need to avoid non-sense expressions like: *isOdd* Tom.
- Then, when defining a function, specify what is the collection of values that can be used as inputs and to which collection of values the produced outputs belong.

We refer to a collection of values as a *type* and we give it a name.

Crunching Values

Who's Your Type

Haskell provides several built-in types such as Bool and Integer.

Obviously, you can define your own ones.

```
data Person = Joe | Tom | Sally
data Color  = Black | Green | Orange | Yellow | Blue | White
```

Type signatures state that a value belongs to a type.

- Joe::Person, which we read as "Joe has type Person".
- $f :: \alpha \to \beta$, i.e. f takes values of type α to values of type β.

Crunching Values

Who's Your Type

Here are the type signatures of all the functions we've defined so far.

$$\begin{array}{llllllllll} \textbf{likesFP} & :: & \text{Person} & \rightarrow & \text{Bool} & \qquad \textbf{eyeColor} & :: & \text{Person} & \rightarrow & \text{Color} \\ \textbf{isGreen} & :: & \text{Color} & \rightarrow & \text{Bool} & \qquad \textbf{isOdd} & :: & \text{Integer} & \rightarrow & \text{Bool} \end{array}$$

In conclusion, to define a function we have to:

- Decide what is the type of the inputs the function will operate on and what is the type of the produced outputs — i.e. establish the function's type signature.
- Come up with a rule to associate each input to a unique output — this rule is defined in terms of equations and often we give a generic way to transform a typical input into an output so to avoid enumerating all possible input/output associations.

Crunching Values

No Country for Old Method

Is a function the same as method in an imperative language? After all, they both take in typed inputs and deliver typed outputs.

Indeed, they are two extremely diverse species:

- A function *always* returns the same output for the same input, a method may not.
- Evaluating a function has no side-effects, whereas a method call may (and often does) change the state of the program.

Crunching Values

No Country for Old Method

Example: we're programming a video game and have a class to track if a character has entered the scene at least once.

```
class Tracker
    Person who;
    bool   seen = false;

    bool seen(Person character)
        if (character == who)   seen = true;
        return seen;
```

Crunching Values

No Country for Old Method

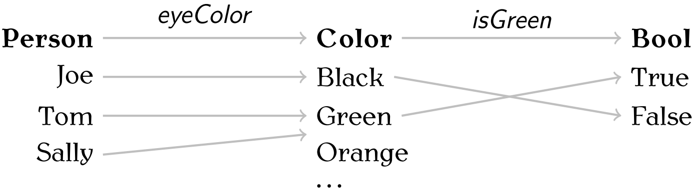
Now look at this code snippet:

```
spotJoe = new Tracker(who = Joe);
// Now:  spotJoe.seen = false

x = spotJoe.seen(Sally);   // x = false
spotJoe.seen(Joe);         // Now:  spotJoe.seen = true
y = spotJoe.seen(Sally);   // y = true

z = x || y;                // z = true
```

Crunching Values

Cooking a Meal

Look at *eyeColor* and *isGreen*, you see that we could take a Person (Tom say) and use *eyeColor* to get his eye Color (Green).



The trick works really for any Person (not just Tom) and so we have a new function.

Crunching Values

Cooking a Meal

Here's how we could define it in Haskell:

```
hasGreenEyes Joe   = isGreen (eyeColor Joe)
hasGreenEyes Tom   = isGreen (eyeColor Tom)
hasGreenEyes Sally = isGreen (eyeColor Sally)
```

$$
\begin{aligned}
&\mathit{hasGreenEyes}\ \text{Joe} \\
&\qquad = \mathit{isGreen}(\mathit{eyeColor}\ \text{Joe}) && \text{by } 1^{st}\ \mathit{hasGreenEyes} \text{ equation} \\
&\qquad = \mathit{isGreen}\ \text{Black} && \text{by } 1^{st}\ \mathit{eyeColor} \text{ equation} \\
&\qquad = \text{False} && \text{by } 2^{nd}\ \mathit{isGreen} \text{ equation.}
\end{aligned}
$$

Crunching Values

Cooking a Meal

Yet another example.

```
f :: Integer → Integer
f x = 2*x + 1

isOddF :: Integer → Bool
isOddF x = isOdd (f x)
```

$$\begin{tikzcd}
\text{Integer} \arrow[d, "f"] & \textbf{Integer} \arrow[dd, "isOddF"] \\
\text{Integer} \arrow[d, "isOdd"] & \\
\text{Bool} & \textbf{Bool}
\end{tikzcd}$$

Crunching Values

Cooking a Meal

$$
\begin{aligned}
&\mathit{isOddF}\ 5 \\
&\quad = \quad \mathit{isOdd}\ (f\ 5) && \text{by substitution} \\
&\quad = \quad \mathit{isOdd}\ (5 * 2 + 1) && \text{by substitution} \\
&\quad = \quad \mathit{isOdd}\ 11 && \text{by arithmetic} \\
&\quad = \quad (11\ `\mathit{mod}`\ 2) == 1 && \text{by substitution} \\
&\quad = \quad 1 == 1 && \text{by } \mathit{mod} \text{ definition} \\
&\quad = \quad \text{True} && \text{by } == \text{ definition.}
\end{aligned}
$$

Crunching Values

Cooking a Meal

Whenever we have two functions

$$f :: \alpha \longrightarrow \beta \qquad g :: \beta \longrightarrow \gamma$$

we can always define a new one, the *composite*

$$\begin{aligned} &h :: \alpha \longrightarrow \gamma \\ &h\ x \ = \ g\ (f\ x) \end{aligned}$$

We can also use the function composition operator: $h = g \circ f$

$$\mathbf{hasGreenEyes} = \mathbf{isGreen} \circ \mathbf{eyeColor}$$

$$\mathbf{isOddF} = \mathbf{isOdd} \circ \mathbf{f}$$

$$\mathrm{pipeline} = \mathbf{f}_n \circ \cdots \circ \mathbf{f}_2 \circ \mathbf{f}_1$$