Functions

Jan van Eijck

May 27, 2003

Abstract

In mathematics, the concept of a function is perhaps even more important than that of a set. Also, functions are crucial in computer programming, as the functional programming paradigm demonstrates. This chapter introduces basic notions and then moves on to special functions, operations on functions, defining equivalences by means of functions, and compatibility of equivalences with operations.

```
module RCRH7

where

import List
```

Functions as a Special Kind of Relations

A **function** is a relation f that satisfies the following condition.

$$(x, y) \in f \wedge (x, z) \in f \implies y = z.$$

That is: for every $x \in \mathrm{dom}\,(f)$ there is *exactly one* $y \in \mathrm{ran}(f)$ such that $(x, y) \in f$.

If $x \in \mathrm{dom}\,(f)$, then $f(x)$ is by definition the unique object $y \in \mathrm{ran}(f)$ for which $(x, y) \in f$.

Functions as Algorithms

The definition abstracts from differences of implementation of functions. Each implementation is based on a particular algorithm for computing values:

```
f1 x = x^2 + 2 * x + 1
g1 x = (x + 1)^2

f1' = \x -> x^2 + 2 * x + 1
g1' = \x -> (x + 1)^2
```

Note that there is (can be) no general way of testing whether two functions are identical (in the set-theoretic sense).

From List of Pairs to Fct, From Fct to List of Pairs

In cases of functions with finite domains it is easy to switch back and forth between the set-theoretic and the computational perspectives, as the following conversions demonstrate.

```
list2fct :: Eq a => [(a,b)] -> a -> b
list2fct [] _ = error "function not total"
list2fct ((u,v):uvs) x | x == u    = v
                       | otherwise = list2fct uvs x

fct2list :: (a -> b) -> [a] -> [(a,b)]
fct2list f xs = [ (x, f x) | x <- xs ]
```

Listing Values

If a function is defined on an enumerable domain, we can list its (finite or infinite) range starting from a given element.

```
listValues  :: Enum a => (a -> b) -> a -> [b]
listValues f i = (f i) : listValues f (succ i)
```

```
RCRH7> take 10 (listValues (2^) 1)
[2,4,8,16,32,64,128,256,512,1024]
RCRH7> take 10 (listValues (^2) 1)
[1,4,9,16,25,36,49,64,81,100]
```

Listing a range

If we also know that the domain is bounded, we can generate the whole function, or the whole range, as a finite list.

```
listF :: (Bounded a, Enum a) => (a -> b) -> [(a,b)]
listF f = [ (i,f i) |  i <- [minBound..maxBound] ]

listRange :: (Bounded a, Enum a) => (a -> b) -> [b]
listRange f = [ f i | i <- [minBound..maxBound] ]
```

```
RCRH7> listF not
[(False,True),(True,False)]
RCRH7> listRange not
[True,False]
```

From . . . to, On, Codomain

Suppose that X and Y are sets. A function f is *from* X *to* Y; notation:

$$f : X \longrightarrow Y,$$

if $\mathrm{dom}\,(f) = X$ and $\mathrm{ran}(f) \subseteq Y$.

In this situation, Y is called the *codomain* of f.

A function f is said to be defined *on* X if $\mathrm{dom}\,(f) = X$.

Note that the set-theoretic way of identifying the function f with the relation $R = \{(x, f(x)) \mid x \in X\}$ has no way of dealing with this situation: it is not possible to recover the intended codomain Y from the relation R. As far as R is concerned, the codomain of f could be any set that extends $\mathrm{ran}(R)$.

Specifying a Function

For completely specifying a function f three things are sufficient:

- Specify $\mathrm{dom}\,(f)$,
- Specify the codomain of f,
- Give an instruction for how to construct $f(x)$ from x.

In Haskell, the first two of these together form the type specification. The third part is taken care of by the set of function equations.

More Than One Argument

Functions can be unary, binary, ternary, and so on. Squaring on $\mathbb{N}$ is unary. Addition and multiplication on $\mathbb{N}$ are binary.

Haskell has predefined operations `curry` and `uncurry` to switch back and forth between functions of types `(a,b) -> c` and `a -> b -> c`.

We can extend this to cases of functions that take triples, quadruples, etc. as arguments.

```
curry3 :: ((a,b,c) -> d) -> a -> b -> c -> d
curry3 f x y z = f (x,y,z)

uncurry3 :: (a -> b -> c -> d) -> (a,b,c) -> d
uncurry3 f (x,y,z) = f x y z
```

Proving that two functions $f, g : X \to Y$ are equal

The general form, spelled out in full, of such a proof is:

Given: $f, g : X \to Y$.
To be proved: $f = g$.
Proof:

Let x be an arbitrary object in X.
To be proved: $f(x) = g(x)$.
Proof:

...

Thus $f(x) = g(x)$.

Thus $f = g$.

Recurrences versus Closed Forms

A definition for a function $f : \mathbb{N} \to A$ in terms of algebraic operations is called a **closed form** definition. A function definition for f in terms of the values of f for smaller arguments is called a **recurrence** for f. The advantage of a closed form definition over a recurrence is that it allows for more efficient computation, since (in general) the computation time of a closed form does not grow exponentially with the size of the argument.

Consider the following recurrence.

```
g 0 = 0
g n = g (n-1) + n
```

A closed form definition of the same function is:

```
g' n = ((n + 1) * n ) / 2
```

Give a closed form implementation of the following function:

```
h 0 = 0
h n = h (n-1) + (2*n)
```

Give a closed form implementation of the following function:

```
k 0 = 0
k n = k (n-1) + (2*n-1)
```

Restriction

Suppose that $f : X \longrightarrow Y$ and $A \subseteq X$. The *restriction* of f *to* A is the function $h : A \to Y$ defined by $h(a) = f(a)$.

The notation for this function is $f \restriction A$

Here is the implementation, for functions implemented as type `a -> b`:

```
restrict :: Eq a => (a -> b) -> [a] -> a -> b
restrict f xs x | elem x xs = f x
                | otherwise = error "not in domain"
```

Image, Co-image

Suppose that $f : X \longrightarrow Y$, $A \subseteq X$ and $B \subseteq Y$.

1. $f[A] = \{f(x) \mid x \in A\}$ is called the *image* of A under f;
2. $f^{-1}[B] = \{x \in X \mid f(x) \in B\}$ is called the co-image of B under f.

From this definition we get:

1. $f[X] = \mathsf{ran}(f)$,
2. $f^{-1}[Y] = \mathrm{dom}\,(f)$,
3. $y \in f[A] \;\Leftrightarrow\; \exists x \in A(y = f(x))$,
4. $x \in f^{-1}[B] \;\Leftrightarrow\; f(x) \in B$.

Here are the implementations of *image* and *co-image*:

```
image :: Eq b => (a -> b) -> [a] -> [b]
image f xs = nub [ f x | x <- xs ]

coImage :: Eq b => (a -> b) -> [a] -> [b] -> [a]
coImage f xs ys = [ x | x <- xs, elem (f x) ys ]
```

This gives:

```
RCRH7> image (*2) [1,2,3]
[2,4,6]
RCRH7> coImage (*2) [1,2,3] [2,3,4]
[1,2]
```

Surjections, Injections, Bijections

A function $f : X \longrightarrow Y$ is called

1. *surjective*, or a *surjection*, or *onto Y*, if every element $b \in Y$ occurs as a function value of *at least* one $a \in X$, i.e., if $f[X] = Y$;
2. *injective*, an *injection*, or *one-to-one*, if every $b \in Y$ is value of *at most* one $a \in X$;
3. *bijective* or a *bijection* if it is both injective and surjective.

Checking that a Function is Injective

If the domain of a function is represented as a list, the injectivity test can be implemented as follows:

```
injective :: Eq b => (a -> b) -> [a] ->  Bool
injective f [] = True
injective f (x:xs) =
   notElem (f x) (image f xs) && injective f xs
```

```
RCRH7> injective (^2) [1..100]
True
RCRH7> injective (even) [1..100]
False
```

Checking that a Function is Surjective

If the domain and codomain of a function are represented as lists, the surjectivity test can be implemented as follows:

```
surjective :: Eq b => (a -> b) -> [a] -> [b] -> Bool
surjective f xs [] = True
surjective f xs (y:ys) =
    elem y (image f xs) && surjective f xs ys
```

```
RCRH7> surjective succ [1..10] [2..11]
True
RCRH7> surjective pred [1..10] [1..10]
False
```

Proving that a Function is Injective/Surjective

The following implication is a useful way of expressing that f is injective:

$$f(x) = f(y) \implies x = y.$$

The contraposition: $x \neq y \implies f(x) \neq f(y)$, of course says the same thing differently.

That $f : X \to Y$ is surjective is expressed by:

$$\forall b \in Y \; \exists a \in X \; f(a) = b.$$

Proof Schemas for Showing Injectivity

To be proved: f is injective.
Proof:
Let x, y be arbitrary, and suppose $f(x) = f(y)$.
$\vdots$
Thus $x = y$.

To be proved: f is injective.
Proof:
Let x, y be arbitrary, and suppose $x \neq y$.
$\vdots$
Thus $f(x) \neq f(y)$.

Proof Schema for Showing Surjectivity

To be proved: $f : X \to Y$ is surjective.
Proof:
Let b be an arbitrary element of Y.
$\vdots$
Thus there is an $a \in X$ with $f(a) = b$.

Function Composition

Suppose that $f : X \longrightarrow Y$ and $g : Y \longrightarrow Z$. Thus, the *codomain* of f coincides with the *domain* of g. The *composition* of f and g is the function $g \circ f : X \longrightarrow Z$ defined by

$$(g \circ f)(x) = g(f(x)).$$

"First, apply f, next, apply g" — thanks to the usual "prefix"-notation for functions, the f and the g are unfortunately in the reverse order in the notation $g \circ f$. To keep this reverse order in mind it is good practice to refer to $g \circ f$ as "g after f".

Function Composition: Implementation

```
(.)             :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x        = f (g x)
```

Example:

```
even n                  = n `rem` 2 == 0
odd                     = not . even
```

Facts About Composition

If $f : X \to Y$, $g : Y \to Z$ and $h : Z \to U$, then $(h \circ g) \circ f = h \circ (g \circ f)$.

This says that composition is *associative*.

Suppose that $f : X \longrightarrow Y$, $g : Y \longrightarrow Z$. Then:

1. $g \circ f$ injective $\Longrightarrow$ f injective,
2. $g \circ f$ surjective $\Longrightarrow$ g surjective,
3. f and g injective $\Longrightarrow$ $g \circ f$ injective,
4. f and g surjective $\Longrightarrow$ $g \circ f$ surjective.

Example Problem

Suppose that $h : X \to X$ satisfies $h \circ h \circ h = 1_X$. Show that h is a bijection.

Given: $h : X \to X$ satisfies $h \circ h \circ h = 1_X$.
To be proved: h is a bijection.
Proof:
To show injectivity, let $a_1, a_2 \in X$ be arbitrary,
and suppose $h(a_1) = h(a_2)$.
Then $h^3(a_1) = h^3(a_2)$, and from the given about h^3 we get $a_1 = a_2$.

To show surjectivity, let $b \in X$ be arbitrary.
From the given about h^3, $h^3(b) = b$. Thus $h(h^2(b)) = b$, so there is an $a \in X$ with $h(a) = b$.

Example Problem, Ctd

Exhibit a simple example of a set X and a function $h : X \to X$ such that $h \circ h \circ h = 1_X$, whereas $h \neq 1_X$.

Take $X = \{0, 1, 2\}$ and $h = \{(0, 1), (1, 2), (2, 0)\}$.

Inverse Function

If we consider $f : X \to Y$ as a relation, then we can consider its relational inverse: the set of all (y, x) with $(x, y) \in f$. However, there is no guarantee that the relational inverse of a function is again a function. In case f is injective, we know that the relational inverse of f is a partial function (some elements in the domain may not have an image). If f is also surjective, we know that the relational inverse of f is a function. Thus, an inverse function of f has to satisfy some special requirements.

Suppose that $f : X \to Y$. A function $g : Y \to X$ is an *inverse* of f if both (i) $g \circ f = 1_X$, and (ii) $f \circ g = 1_Y$.

1. A function has at most one inverse.
2. A function has an inverse iff it is bijective.

Examples of Functions with Inverses

The real function f that is given by $f(x) = \frac{9}{5}x + 32$ allows us to convert degrees Celcius into degrees Fahrenheit. The inverse function f^{-1} is given by $f^{-1}(x) = \frac{5}{9}(x - 32)$; it converts degrees Fahrenheit back into degrees Celsius.

Here are integer approximations:

```
c2f, f2c :: Int -> Int
c2f x = div (9 * x) 5 + 32
f2c x = div (5 * (x - 32)) 9
```

Examples of Functions with Inverses –Ctd

The class *Enum* is defined in Haskell as follows:

```
class Enum a where
    succ, pred          :: a -> a
    toEnum              :: Int -> a
    fromEnum            :: a -> Int
```

`fromEnum` should be a *left-inverse* of `toEnum`:

```
fromEnum (toEnum x) = x
```

This requirement cannot be expressed in Haskell, so it is the responsibility of the programmer to make sure that it is satisfied.

Examples of use of `toEnum` and `fromEnum`

```
ord                    :: Char -> Int
ord                     = fromEnum
chr                    :: Int -> Char
chr                     = toEnum
```

Partial Functions

A partial function from X to Y is a function with its domain included in X and its range included in Y. If f is a partial function from X to Y we write this as $f : X \hookrightarrow Y$. It is immediate from this definition that $f : X \hookrightarrow Y$ iff $\mathrm{dom}\,(f) \subseteq X$ and $f \restriction \mathrm{dom}\,(f) : \mathrm{dom}\,(f) \to Y$.

A way of defining a partial function:

$$f(x) = \begin{cases} \bot & \text{if ...} \\ t & \text{otherwise} \end{cases}$$

The computational importance of partial functions is in the systematic perspective they provide on exception handling. In Haskell, the crude way to deal with exceptions is by a call to the error abortion function `error`.

The code below implements partial functions `succ0` and `succ1`.

```
succ0 :: Integer -> Integer
succ0 (x+1) = x + 2

succ1 :: Integer -> Integer
succ1 = \ x -> if x < 0
               then error "arg out of range"
               else x+1
```

The disadvantage of these implementations is that if `succ0` or `succ1` is called by another program, the execution of that other program may abort.

A useful technique for implementing partial functions is to represent a partial function from type a to type b as a function of type `a -> [b]`. In case of an exception, the empty list is returned. If a regular value is computed, the unit list with the computed value is returned.

```
succ2 :: Integer -> [Integer]
succ2 = \ x -> if x < 0 then [] else [x+1]
```

Composition of partial functions implemented with unit lists can be defined as follows:

```
pcomp :: (b -> [c]) -> (a -> [b]) -> a -> [c]
pcomp g f = \ x -> concat [ g y | y <- f x ]
```

```
RCRH7> (pcomp succ2 succ2) 2
[4]
```

As an alternative to this trick with unit lists Haskell has a special datatype for implementing partial functions, the datatype `Maybe`, which is predefined as follows.

```
data Maybe a = Nothing | Just a
               deriving (Eq, Ord, Read, Show)

maybe              :: b -> (a -> b) -> Maybe a -> b
maybe n f Nothing  = n
maybe n f (Just x) = f x
```

Here is a third implementation of the partial successor function:

```
succ3 :: Integer -> Maybe Integer
succ3 = \ x -> if x < 0 then Nothing else Just (x+1)
```

The use of the predefined function `maybe` is demonstrated in the definition of composition for functions of type `a -> Maybe b`.

```
mcomp :: (b -> Maybe c) ->
            (a -> Maybe b) -> a -> Maybe c
mcomp g f = (maybe Nothing g) . f
```

```
RCRH7> (mcomp succ3 succ3) 2
Just 4
```

Dealing with Exceptions

The `maybe` function allows for all kinds of ways to deal with exceptions. E.g., a function of type `a -> Maybe b` can be turned into a function of type `a -> b` by the following `part2error` conversion.

```
part2error :: (a -> Maybe b) -> a -> b
part2error f =
   (maybe (error "value undefined") id) . f
```

```
RCRH7> succ3 0
Just 1
RCRH7> part2error succ3 0
1
RCRH7> succ3 (-1)
Nothing
RCRH7> part2error succ3 (-1)

Program error: value undefined
```

Functions as Partitions

Here is a Haskell implementation of a procedure that maps a function to the equivalence relation inducing the partition that corresponds with the function:

```
fct2equiv :: Eq a => (b -> a) -> b -> b -> Bool
fct2equiv f x y = (f x) == (f y)
```

You can use this to test equality modulo n, as follows:

```
Main> fct2equiv (`rem` 3) 2 14
True
```

Equivalence Classes Defined by Functions

Equivalence classes (restricted to a list) for an equivalence defined by a function are generated by the following Haskell function:

```
block :: Eq b => (a -> b) -> a -> [a] -> [a]
block f x list = [ y | y <- list, f x == f y ]
```

This gives:

```
RCRH7> block (`rem` 3) 2 [1..20]
[2,5,8,11,14,17,20]
RCRH7> block (`rem` 7) 4 [1..20]
[4,11,18]
```

Congruences

A function $f : X^n \to X$ is called an n-ary operation on X. Addition and multiplication are binary operations on $\mathbb{N}$ (on $\mathbb{Z}$, on $\mathbb{Q}$, on $\mathbb{R}$, on $\mathbb{C}$).

If one wants to define new structures from old, an important method is taking quotients for equivalences that are compatible with certain operations.

If f be an n-ary operation on A, and R an equivalence on A, then R is a **congruence** for f (or: R is **compatible** with f) if for all $x_1, \ldots, x_n, y_1, \ldots, y_n \in A$:

$x_1 R y_1, \ldots, x_n R y_n$ imply that $f(x_1, \ldots, x_n) R f(y_1, \ldots, y_n)$.

If R is a congruence for f, then the operation induced by f on A/R is the operation $f_R : (A/R)^n \to A/R$ given by

$$f_R(|a_1|_R, \ldots, |a_n|_R) := |f(a_1, \ldots, a_n)|_R.$$

If (A, f) is a set with an operation f on it and R is a congruence for f, then $(A/R, f_R)$ is the *quotient structure* defined by R.

Examples

We will show that $(\mathrm{mod}\ n)$ is a congruence for multiplication.

Recall the definition of $(\mathrm{mod}\ n)$: $m \equiv k\ (\mathrm{mod}\ n)$ iff m and k have the same remainder when divided by n.

More formally: $m \equiv k\ (\mathrm{mod}\ n)$ (or: $m \equiv_n k$) iff

- $m = qn + r$, with $0 \leq r < n$,
- $k = q'n + r'$, with $0 \leq r' < n$,
- $r = r'$.

The following gives a convenient way to test equivalence modulo n:

Proposition 1 $m \equiv_n k$ *iff* $n \mid m - k$.

Proof.

$\Rightarrow$: Suppose $m \equiv_n k$. Then $m = qn + r$ and $k = q'n + r'$ with $0 \leq r < n$ and $0 \leq r' < n$ and $r = r'$. Thus, $m - k = (q - q')n$, and it follows that $n \mid m - k$.

$\Leftarrow$: Suppose $n \mid m - k$. Then $n|(qn + r) - (q'n + r')$, so $n \mid r - r'$. Since $-n < r - r' < n$, this implies $r - r' = 0$, so $r = r'$. It follows that $m \equiv_n k$. $\square$

The following are equivalent:

- $m \equiv_n k$.
- $n \mid m - k$.
- $\exists a \in \mathbb{Z} : an = m - k$.
- $\exists a \in \mathbb{Z} : m = k + an$.
- $\exists a \in \mathbb{Z} : k = m + an$.

Suppose $m \equiv_n m'$ and $k \equiv_n k'$.

We have to show:

$m \cdot k \equiv_n m' \cdot k'$.

From $m \equiv_n m'$ we get that there is an $a \in \mathbb{Z}$ with $m' = m + an$.

From $k \equiv_n k'$ we get that there is a $b \in \mathbb{Z}$ with $k' = k + bn$.

Therefore $m'k' = mk+akn+bmn+abn^2 = mk+(ak+bm+abn)n \equiv_n mk$.

It follows that we can define:

$$[m]_n \cdot [k]_n := [m \cdot k]_n.$$

Questions

Is $(\mathrm{mod}\ n)$ a congruence for addition?

In other words: is it possible to define addition of classes in $\mathbb{Z}_n$ by means of:

$$[k]_n + [m]_n := [k+m]_n$$

Is $(\mathrm{mod}\ n)$ a congruence for subtraction?

In other words: is it possible to define subtraction of classes in $\mathbb{Z}_n$ by means of:

$$[k]_n - [m]_n := [k - m]_n$$

Is $(\mathrm{mod}\ n)$ a congruence for exponentiation? In other words: is it possible to define exponentiation of classes in $\mathbb{Z}_n$ by means of:

$$([k]_n)^{([m]_n)} := [k^m]_n, \text{ for } k \in \mathbb{Z}, m \in \mathbb{N}.$$