Curry programming language

Santiago Suárez Pérez

EAFIT University

November 30th 2015

Contents

Introduction

- Curry is a "Truly Integrated Functional Logic Language"
- Actually still on development.
- Uses PAKCS compiler, created by Michael Hanus.
- Embraces narrowing and residuation (explanation later).

Main Features

Curry is mainly based on Haskell sintax, implementing the concepts of logic programming and creating new ones for the functional-logic paradigm.
In curry you can find features of functional programming, such as high-order functions and nested expressions. And also logic programming features, like logical variables, partial data structures, etc.

Must notice

- The PAKCS compiler only works on SWI-Prolog 6.x or SICSTUS-Prolog.
- Pretty much seamly to Haskell REPL.
- Use the comand "`Prelude> :h`" so you can see help.
- For loading a program, use "`Prelude> :l program_file`", notice you don't need to put the file extension.
- You can also evaluate math expressions, like "`Prelude> 3 * 4`".

Installing on Linux

Download the *.tar.gz* from the Portland Aachen Kiel Curry System webpage. Then, run the `make` command from bash, and then add the `packsx.x.x/bin/` folder to your `PATH` environment variable.
After this, you can just run the command `"$curry"` or `"$pakcs"`, and you'll get into the Curry REPL.
Also you need to install the **ghc** Haskell Compiler before installing PAKCS.

Free variables

In order to curry instantiating variables with all posibilities, you need to free the variables. For example, if we want to know the posibilities of the logic expression:

```
Prelude> x && (y || (not x))
```

The PAKCS compiler will show an error since the variables are not instanciated to nothing. So you must write:

```
Prelude> x && (y || (not x)) where x,y free
```

Also, you can activate the `free mode` of the REPL using "`:set +free`". Freeing variables means that compiler will use any valor for them and will show all possible results. This is mainly used for logic programming.

Declaring facts on curry

In logic programming, you must declare facts. Facts are like a database for your program. Unlike Prolog, in Curry you need to define your constants, so you cannot declare constants from nowhere since Curry is strongly-typed, and Prolog is a-typed.
For example, let's declare a family organization.

The data

```
data Person = Christine | Maria | Monica | Alice | Susan |
Anthony | Bill | John | Frank | Peter
```

Now let's declare women and men.

```
female Christine = success
female Maria = success
male Anthony = success
...
```

Facts on Curry... II

You can also declare facts between two or more constants, so you stablish a relationship. For example:

The 'married' relationship

```
married Christine Anthony = success
married Maria Bill = success
...
```

The mother relationship

```
mother Christine John = success
mother Maria Fran = success
mother Monica Peter
...
```

Clauses on Curry

The actual motion for declaring clauses on curry is pretty much simple. You just need to free your variables using the **where** or **let** clauses (explanation later), and the add the logical operation which is ussually '&', since logic programming uses mostly disyuntive normal form. For example, given the above facts, let's define the Father and GrandFather clauses.

The Father clause

```
father f c = let m free in married m f & mother m c
```

The Grandfather clause

```
grandfather f c = let f free in father g f & father f c
grandfather f c = let f free in mother g m & mother m c
```

An example with graphs

We all know as programmers what a graph is. So supose you have a database with the edges of the graph, and you want to know if there's a path from a node to another.
You can easily define the `path` clause.

The path clause

```
path a z = edge a b && path b z where b free
```

And this is how you can program graphs algorithms with logic programming in Curry. Notice that freeing the `b` variable, your letting it to take all possible values in your database.

& vs &&

&&

The && operator is used to evaluate `Boolean` expressions. If you apply this operator, the compiler will evaluate both operands and if both evaluate to "True" the this will return True.

&

This operator is mostly used for logic evaluations, if we apply this to `u & v`, this will evaluate both `u` and `v`, and if both succeed, then this will succeed.

Functions in Curry

Curry is very has a Haskell-like syntax, so you can define functions very easily.

Examples

```
cube x = x*x*x
result = (\x -> x * x) (2+3)
```

Anonymous functions don't need a name, and are written as above.

Conditions

Defining conditions in Curry is almost the same way in Haskell. Just define explicit cases.

Example 2

```
max x y | x < y = y
| otherwise = x
```

The `otherwise` keyword is used as an `else` for another languages.

Where clauses

The where clauses let you ingress an expression inside other, and is easy to read.
The main structure is to define an expression where the functions can or cannot be defined previously, and then define the undefined functions.
The where clause is mainly used in expressions that need the variables to be free, as explained before.

Example 3: As a formula

```
heron a b c = sqrt (s * (s - a) * (s - b) * (s - c))
where s = (a + b + c)/2
```

So, if you're using results of previous formulas in your math calculations, you can use the where clause.

Let clauses

Let clauses act almost the same as where clauses, which allow you to define a new expression inside the expression. You can also define multiple expressions in a let clause (and also with the where clause).

Zipping with let clause

```
zipp = let
    f x = g x+1 ; g x = x + 2
    in zip l (map f l)
```

Let vs Where

In functional programming there are many discutions about the uses of the where and the let clause, but the most important thing is to notice that the let clause creates a scope **before** the expression, and the where clause creates a scope **after** the expression.

High-order functions

High-order functions are functions which take another function as a parametter. In Curry, sintax looks like Haskell for defining this type of functions.

Example 4

```
map f (x:xs) = (f x) :  (map f xs)
applyTwice f x = f (f x)
filter bool_func (x:xs)
  | bool_func x = x :  filter xs
  | otherwise = filter xs
```

Residuation

Evaluating expressions in Curry is done by two ways: Narrowing and Residuating.
Supose you have and expression and a variable on it. If you're residuating, you need to know the value of the variable that is ocurring on the expression, so if the variable doesn't have any value then the evaluation connot be done.
However, if there might be an expression which can assign a value to the variable, the this will be computed and then the evaluation of the first expression will resume. Otherwise, the evaluation fails.
For example, the Boolean equality (==) returns a Boolean value, and it's computed with Residuation. Execute this:

```
z==2+2 where z free
```

Narrowing

Narrowing is like the opposite of residuation, since it guesses a value for the variable mentioned just earlier.
The guessed values are not shown until these values make possible the computation of the expression.
Following the example given, Constrained equality (=:=) is different from Boolean equality since it returns a **Success** type, and also is computed by narrowing.
Try to execute this:

```
z=:=2+2 where z free
```

Flexible & Rigid

The operations that residuate are called *rigid*, and the ones that narrow are called *flexible*.
Mostly defined operations are flexible, but the arithmetic operations are rigid since guessing is not a reasonable option for them.
For example, the list concatenation (++) is a flexible operation, so you can evaluate:

```
x ++ [3,4] =:= [1,2,3,4] where x free
```

On the other hand, predefined math operations like addition are rigid, so calling addition with logical variables like this...
`x+2 =:= 4 where x free`, what do you think it will show?

Important Note

The flex or rigid state of a function doesn't make no difference for expressions without logical variables.

Example programs

- Family relationships
- Natural numbers operations
- Mergesort