Curry
A truly integrated functional logic language

Santiago Palacio Gómez

Universidad EAFIT

29 de mayo de 2013

I do not take credit for the content here. All the research on this matter was made by Michael Hanus and partners. I only intend to make a summary, of what i think are the most important aspects of the language; all with educational purposes.

Table of Contents I

Table of Contents II

Curry is a programming language designed to join the most important concepts from declarative programming paradigms. It combines features from functional programming (nested expressions, lazy evaluation, high-order functions,...) and logic programming (logic variables, built-in search,...).

Table of Contents I

Table of Contents II

An expression is either an *atom* (literal or symbol) or an application of an expression to another expression. For example, the expressions "`2`" or "`True`" are considered *atoms*, while "`2+3`" or "`not True`" are complex expressions; those combinations are referred to as *function application*. In curry, like in many other functional languages, function application is written simply by juxtaposition of terms with spaces in between. The result of evaluating an expression is a value, e.g. `2+3` has a value of `5`.

Curry provides functions as a *procedural abstraction*. Functions can be viewed as a parametrized expression, possibly with a name. Unlike with pure functional languages, in curry functions are non deterministic. This is, same parameters may result in different return values. This is due to the integration with the logic paradigm.

In the previous examples, `+` and `not` were functions, both defined in the *prelude*. We see that curry provides infix operators (functions) so we can write normal aritmethic expressions like `2+5*6`, and also provides operator precedence and asociativiry so `1+2+5*6` is interpreted like `(1+2)+(5*6)`. For example, in curry, some arithmetic operators can be defined like:

```
infixl 7 *, `div`, `mod`
infixl 6 +, -
infix 4 <, >, <=, >=
```

Curry provides some basic types, referred as *builtin types*. Here are some examples

- **Int** Integers with arbitrary precision.
- **Bool** Booleans, can take only `True` or `False` values.
- **Char** Characters from ascii.
- **$[\tau]$** Lists that are either empty `[]` or a concatenation of an element of type τ with another list `x:list`.
- **String** Strings are seen in curry as a list of characters.
- **()** Unit, used when the return value of a function is not relevant.

Table of Contents I

Table of Contents II

In curry, functions are values themselves. This is, the parameter of a function, the result of a function, the result of an expression can be a function itself.
Functions that take other functions as parameters are called *High-Order Functions*.
For example, a function comonly predefined in most functional languages, `map` is a function that takes another function, and applies it to the elements of a list.
It might be defined as follows:

```
map _ [] = []
map f (x:xs) = (f x):(map f xs)
```

Functions, being values, have types too. In mathematics we define a function like this:

$$f : A \mapsto B$$

$$x \rightarrow f(x)$$

In Curry, like in Haskell, functions type are defined this way:

```
f :: a -> b
```

where a is the type of the first parameter, and b is the result. Map, for example, has this type

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

The operator `->` is right associative, i.e. `a -> b -> c = a -> (b -> c)`. This is because Curry uses currying. So every function in curry actually takes just one parameter and returns another function that takes the rest of the parameters.
Using this, we could write expressions like:

```
map ((+) 3)
```

And the result would be a function that adds 3 to each element from a list of Integers.

Table of Contents I

Table of Contents II

The scope of an identifier of a function, variable, type, etc. is where it can be referenced in the program. For example, in a function definition, the identifiers of the parameters are available at the whole expression, so in the expressions

```
square x = x*x
cube x = x*x*x
```

altough the parameter is named in the same way in both functions, they are completely separated. Curry is a statically scope language. So the scope of an identifier depends on the program and not on the execution. There are some ways to limit the scope of an identifier.

A `where` creates a scope inside an expression.

```
zipp1 l = zip l (map f l)
          where f x = x+1
```

Here, the function `f` can only be called inside `zipp1`, if at any other point in the program we tried to use this `f` we would get an error.

In general, we can write a `where` expression by doing:

```
e1 where
   e2
   e3...
```

And the identifiers that the expressions at the right of the where expression creates can be used on both sides of the expression.

Let clauses are a way to define identifiers *before* the scope.

```
zipp2 l = let
    f x = g x; g x = x+2
  in
   zip l (map f l)
```

Here the structure of a general `let` expression is:

```
let e1 ; e2..
in
  e3
```

where the bindings created in `e1`, `e2`,... can be used in the whole expression.

Table of Contents I

Table of Contents II

From the definitions in pure functional programming, we borrow the definitions of functions, constructors, patterns and TRS (*term rewiriting system*). It's worth remebering the definition of a TRS as a set of rewriting rules of the form $l \to r$ with linear pattern l as lhs and a term r as rhs.
We must notice that we have changed the traditional definition of the TRS by not requiring that $var(l) \subseteq var(r)$.

We will also take the definitions of position p in a term t ($t|_p$), term replacement $t[s]_p$ and substitutions.
A term t is called *irreducible* or in *normal form* if there is no term s such that $t \rightarrow s$.

The goal of a sequence of rewrite steps is to compute a normal form. A *rewrite strategy* determines for each step a rule and a position to apply the next step. A *normalizing strategy* is one that terminates a rewrite sequence in a normal form, when it exists.

Sometimes, the result itself could not be important. For example take the function

```
idNil [] = []
```

If we try to find the normal form of `idNil[1+2]` we would get `idNil[3]` (note that in Haskell, we would get an error).
So, the interesting results of functional computations are *constructor terms* or *values*.

Functional logic languages are able to do more than pure functional languages since they instantiate variables in a term (free variables) in order to apply the rewrite step. The combination of variable instantiation and rewriting is called **narrowing**.

Formally, $t \rightsquigarrow_{p,R,\sigma} t'$ is a *narrowing step* if $t|_p$ is not a variable, and $\sigma(t) \rightarrow_{p,R} t'$.
Since the substitution σ is intented to instantiate the variables in t, we can restrict $Dom(\sigma) \subseteq Var(t)$. Since in functional logic languages we are interested in computing values, as well as answers, we say that $t \rightsquigarrow^{*}_{\sigma} c$ computes the value c with answer σ if c is a value.

Consider the following program, containing the definition of naturals, the add operation and a “less than or equal” test.

```
data Nat = 0 | S Nat

add 0 y = y
add (S x) y = S (add x y)

leq 0 _ = True
leq (S _) 0 = False
leq (S x) (S y) = leq x y
```

Now, consider the initial term `leq v (add w 0)` where v and w are free variables. By applying leq_1, v is instantiated to 0 and the result `True` is computed:

$$\texttt{leq v (add w 0)} \rightsquigarrow_{\{v \mapsto 0\}} \texttt{True}$$

However, we could also do the following:

$$\texttt{leq v (add w 0)} \rightsquigarrow_{\{w \mapsto 0\}} \texttt{leq v 0} \rightsquigarrow_{\{v \mapsto 0\}} \texttt{True}$$

But this would not be optimal since it computes the same value as the first derivation with a less general answer.

Table of Contents I

Table of Contents II

Needed Narrowing is based on the idea to perfomr only narrowing steps that are necessary to compute a result. This kind of strrategies are also called *lazy* or *demand-driven*.
If there is an argument position that is constructor rooted, then the corresponding actual argument must also be evaluated (or non-deterministically instantiated if it's a variabe) to be constructor rooted.

Consider again the program of Natural numbers. Needed narrowing instantiates the variable `v` in `leq v (add w 0)` to either `0` or `S z` (where z is a fresh variable). In the first case, only rule leq_1 become applicable. In the scond case, only rules leq_2 or leq_3 become applicable. Since the latter rules have a constructor-rooted term as second argument, the corresponding subterm `add w 0` is recursively evaluated to a constructor-rooted term.

Since not every TRS allows such reasoning, needed narrowing is defined on the subclass of *inductively sequential TRS*. We will consider only the lhs of the rules, since they are the only important part for the applicability of needed narrowing. We will characterize a *definitional tree* T (using the *subsumption ordering*: $t \leqslant \sigma(t)$) of an operation f with the following properties:

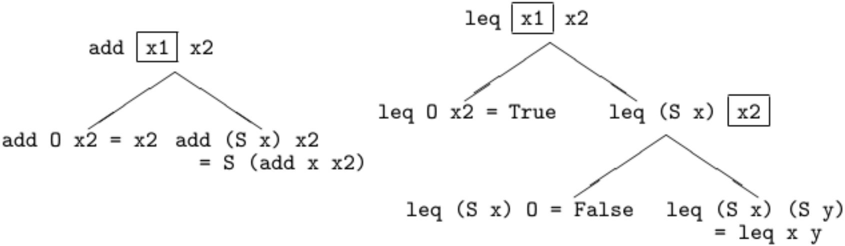
Leaves property
: The maximal elements of T, called the *leaves*, are the lhs of the rules defining f.

Root property
: Thas a minimum element, called the *root*, of the form $f(x_1, x_2, \ldots, x_n)$ where $x_1, \ldots x_n$ are pairwise distinct variables.

Parent property For every pattern π different from the root, there exists a unique π' such that $\pi' < \pi$ and there isn't any π'' such that $\pi' < \pi'' < \pi$.

Induction property Every child of π differs from each other only at a common position, called the *inductive position*, which is the position of a variable in π.

An operation is called inductively sequential if it has a definitional tree and its rules do not contain extra variables. A TRS is inductively sequential if every define operation is inductively sequential. Needed narrowing is applicable to most operations in logic functional languages (and every operation in pure functional languages), however extensions may be useful for particular operations.


add x1 x2
add 0 x2 = x2
add (S x) x2
= S (add x x2)
leq x1 x2
leq 0 x2 = True
leq (S x) x2
leq (S x) 0 = False
leq (S x) (S y)
= leq x y

Definitional Trees are particulary useful because they can be computed at compile time and they contain all infromation for the decisions to the steps in the rewriting process.
We could define a needed narrowing step as an application to an operation-rooted term t by considering it's definitional tree from the root. The tree is recursively processed until one finds a *maximal* pattern that *unifies* with t (similarly with the M.G.U finding process in logic languages). From there (with the new tree) we perform the operation (at each node π as it follows:

If π is a leaf we apply the corresponding rule

If π is a branch let p be it's inductive position, we consider the corresponding subterm$t|_p$

- If $t|_p$ is rooted by a constructor c, if there is a child with c at the inductive position, we examine the child, else we fail.
- If $t|_p$ is a variable, we nondeterministaclly instantiate this variable by the constructor term at the inductive position of a child, and proceed to examine the child.
- If $t|_p$ is operation rooted, we recusively apply the computation of a needed narrowing step to $\sigma(t|_p)$, where σ is the substution, result of previous case distinctions.

The equality symbol `=:=` is called *strict equality*, i.e. te equation $t_1 =:= t_2$ is satisfied iff t_1 and t_2 are reducible to the same *ground* constructor or term. (Note that when t_1 is not reducible, $t_1 =:= t_1$ does not succeed).
We can define $=:=$ as follows:

$$
\begin{array}{lll}
c =:= c & = \texttt{Success} & \forall c/0 \\
c x_1 \ldots\ x_n =:= c y_1 \ldots\ y_n & = x_1 =:= y_1\ \&\ \ldots x_n =:= y_n & \forall c/n \\
\texttt{Success}\ \&\ \texttt{Success} & = \texttt{Success} &
\end{array}
$$

A solution for an equation $t_1 =:= t_2$ is a substitution σ, if $\sigma(t_1) =:= \sigma(t_2) \rightsquigarrow^* \texttt{Success}$.
We have then, that *needed narrowing* is Correct, Complete and Minimal (if there are two derivations, then their substitutions are independent). And, in successful derivations, needed narrowing computes the *shortest* of al possible narrowing derivations.x

If we take the code:

```
or _ True = True
or True _ = True
or False False = False
```

We can see that the rule or doesn't have a definitional tree in the sense that when one of the arguments is True, it is not clear which rule should be evaluated. So we can extend the definition of *inductivel sequential TRS* to a *weakly orthogonal TRS* by requiring only that, for all variants of rules $l_1 \rightarrow r_1$, $l_2 \rightarrow r_2$, if $\sigma(l_1) = \sigma(l_2)$ then $\sigma(r_1) = \sigma(r_2)$.

Then, we can also extend the definition of *definitional trees* by adding or-branches, which are conceptually the union of two definitional trees.
In the previous example, we could create a tree for the rules or_2, or_3 and the rule or_1, then we could join those trees by an or-branch.
This new way of resolving operations is also confluent, for the condition we required.

This same principe may also be extended to handle non-deterministic operations and extra variables, by simply examining every possible or branch, and not requiring that all the rules are confluent to the same normal form. For example, the rule

```
x ? _ = x
_ ? y = y
```

Gives two results for `0 ? 1`, namely `0` and `1`.

Table of Contents I

Table of Contents II

- Curry: A tutorial Introduction. Draft of December 2007. Antoy Sergio, Hanus Michael, Taken from http://www.informatik.uni-kiel.de/ curry/tutorial/ the 15th of April, 2013.
- Functional Logic Programming: From Theory to Curry. Hanus Michale. Institut für Informatik, CAU Kiel, Germany. As sent May the 13th of 2013.