Curry
Integrating Logic and Functional Paradigms

Santiago Palacio-Gómez

Universidad EAFIT
Seminario de Lógica y Computación

September 11, 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

Curry related first paper was published in 1995, by Michael Hanus, Herber Kuchen and Juan José Moreno-Navarro, for the International Logic Programming Symposium in 1995, Workshop on Visions for the Future of Logic Programming.

Table of Contents I

Main Elements

From functional languages, Curry takes elements such as:

- Expressions.
- Functions, high order functions.
- Types.
- Scope (where clauses, let clauses).

Table of Contents I

Basic Definitions

From the definitions in pure functional programming, we borrow:

- Function definition.
- Constructors.
- Patterns
- TRS (*term rewriting system*).

TRS

It's worth remembering the definition of a TRS as a set of rewriting rules of the form $l \rightarrow r$ with linear pattern l as lhs and a term r as rhs.
It must also be noticed, that the traditional definition of the TRS was changed, by not requiring that $var(l) \subseteq var(r)$.

Example

```
add Z y = y
add (S x) y = S (add x y)
```

Basic Definitions II

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 $t \to s$.

Rewrite Strategy

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.

Constructor Rooted Normalized Form

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

Example

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

If we try to find the normal form of `idNil[1+2]`, we would get `idNil[3]` as normal form.

So, the interesting results of functional computations are *constructor terms* or *values*.

Narrowing

Functional logic languages are more flexible 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**.

Example

```
last :: [a] -> a
last x
     | _++[e] =:= x = e
```

Formal Definition

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 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.

Example

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

Example

```
data Nat = Z | S Nat

add Z y = y
add (S x) y = S (add x y)

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

Efficiency

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

$$\texttt{leq v (add w Z)} \leadsto_{\{v \mapsto Z\}} \texttt{True}$$

However, we could also do the following:

$$\texttt{leq v (add w Z)} \leadsto_{\{w \mapsto Z\}} \texttt{leq v Z} \leadsto_{\{v \mapsto Z\}} \texttt{True}$$

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

Definition

- Designed to perform only necessary narrowing steps.
- *Lazy* or *demand-driven*.

When performing a narrowing step, if an argument expression must be constructor rooted:

- If the corresponding position is a variable, it's non-deterministically instantiated.
- If the corresponding position is an expression, it's evaluated to be constructor-rooted.

Example

Consider again the program of Natural numbers and the expression `leq v (add w Z)`.

To get every possible result:

- How is `v` instantiated?
- How is `w` instantiated?

Inductively Sequential TRS

To simplify the computational process of needed narrowing, Inductively Sequential TRS are defined. We will characterize a *definitional tree* T (using the *subsumption ordering*: $t \leqslant \sigma(t)$) of an operation f with the following properties. Each property will be exemplified from using the natural numbers example.

Leaves Property

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

Example

The leaves of `add` are `add Z y` and `add (S x) y`.

Root Property

Has 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.

Example

The root of `add` is `add x y`.

Parent Property

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

Example

`leq x y` could* be parent of `leq (S x') y` and `leq x (S x')`; however, it is not the parent of `leq (S x') (S y')`.

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.

Example

`leq (S x) y` and `leq Z y` could be siblings; however `leq (S x) y` and `leq x (S y)`, differ in two positions, thus could not be siblings.

Inductively Sequential TRS II

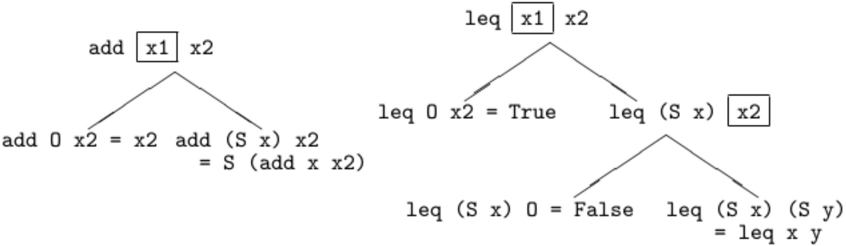
- An operation is called inductively sequential, if it has a definitional tree and its rules do not contain extra variables.
- A TRS is inductively sequential, if every define operation is inductively sequential.

Applicability

Needed narrowing is applicable to many operations in logic functional languages (and every operation in pure functional languages), however extensions may be useful for particular operations.

Example

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


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

Use of Definitional Trees in Narrowing

Definitional Trees can be computed at compile time and they contain all information 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. First, we find the *maximal* node pi that unifies with t, and applies the following algorithm.

Needed Narrowing Algorithm

- 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 non-deterministaclly 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 substitution, result of previous case distinctions.

Strict Equality

The equality symbol `=:=` is called *strict equality*, i.e. the 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:

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

Strict Equality Solutions

A solution for an equation $t_1 =:= t_2$ is a substitution σ, if $\sigma(t_1) =:= \sigma(t_2) \leadsto^* \texttt{Success}$.
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 all possible narrowing derivations.

Weakly Needed Narrowing

If we take the previously shown code:

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

We must extend the definition of *inductively sequential TRS* to a *weakly orthogonal TRS* by requiring only that, for all variants of rules $l_1 \to r_1$, $l_2 \to r_2$, if $\sigma(l_1) = \sigma(l_2)$ then $\sigma(r_1) = \sigma(r_2)$.

Weakly Needed Narrowing II

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.

Table of Contents I

Handling Non-Determinism

This same principle 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`.

Lazy vs Strict Evaluation

Consider the following functions

```
choose x _ = x
choose _ y = y

coin = choose 0 1

double x = x+x
```

What would happen if we call `double coin`?.

Lazy vs Strict Evaluation Examples

Example

```
coin = 0 ? 1

double x = x+x



insert e [] = [e]
insert e (x:xs) = (e:x:xs) ? (x : (insert e xs))

perm [] = []
perm (x:xs) = insert x (perm xs)
```

Lazy vs Strict Evaluation Examples II

Example

```
sorted :: [Integer] -> [Integer]
sorted [] = []
sorted [x] = [x]
sorted (x:y:xs) | ((x < y)::Bool) =:= True
                = x : (sorted (y:xs))

mySort x = sorted (perm x)
```

Table of Contents I

Bibliography

- 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 Michael. Institut für Informatik, CAU Kiel, Germany. As sent May the 13th of 2013.
- Curry An integrated functional language. September 11,2012. Hanus Michael, retrieved the 15th of April 2013.
- Curry: A truly Integrated Functional Logic Language. December 1995. Hanus Michael, Kuchen Herbert, Moreno-Navarro Juan José, retrieved 9th of September 2013.