Two Brief Notes
Concerning Lectures T-09 and T-10

Algebra of Programming Course
CP/1011.8204N5, 2nd year, LEI, U.Minho

J.N.Oliveira

April 2011

"Good methods, properly explained, sell themselves."
David Parnas [3]

Note 1 — What is the meaning of `curry`?

Haskell Prelude:

$$curry :: ((a, b) \to c) \to (a \to b \to c)$$
$$curry\ f\ a\ b = f\ (a, b)$$

Looking closer:

$$\overbrace{(\underbrace{curry\ f}_{\bar{f}}\ a)}^{g}\ b \quad = \quad f\ (a, b)$$

Turning function application explicit, as in `Cp.hs`,

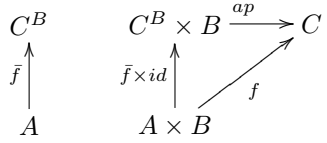
$$ap :: (a \to b, a) \to b$$
$$ap\ (f, a) = f\ a$$

we calculate:

$$\begin{array}{ll} & g\ b\ =\ f(a, b) \\ \equiv & \quad \{ \text{ since } g\ b = ap(g, b) \ \} \\ & ap(g, b)\ =\ f(a, b) \\ \equiv & \quad \{ \text{ since } g = \bar{f}\ a \text{ ; identity function } \} \end{array}$$

$$
\begin{array}{ll}
 & ap(\bar{f}\ a, id\ b)\ =\ f(a,b) \\
\equiv & \qquad \{\ \text{product of functions: } (f \times g)(x,y) = (f\ x, g\ y)\ \} \\
 & ap((\bar{f} \times id)(a,b))\ =\ f(a,b) \\
\equiv & \qquad \{\ \text{composition}\ \} \\
 & (ap \cdot (\bar{f} \times id))(a,b)\ =\ f(a,b) \\
\equiv & \qquad \{\ \text{extensional equality (=remove points)}\ \} \\
 & ap \cdot (\bar{f} \times id)\ =\ f
\end{array}
$$

In a diagram, denoting type $B \to C$ by C^B:



This means that $\bar{f}$ is a solution of the equation $ap \cdot (k \times id)\ =\ f$:

$$k = \bar{f} \quad \Rightarrow \quad ap \cdot (k \times id)\ =\ f$$

It turns out to be the **unique** such solution:

$$k = \bar{f} \quad \Leftrightarrow \quad ap \cdot (k \times id)\ =\ f$$

Thus we have a universal property. The follow up of this can be found in chapter 3 of [2].

Note 2 — Where do algorithms come from?

Where do algorithms come from? From human imagination only? Surely not — they actually emerge from mathematics. This note provides a naive introduction to algorithm synthesis by showing how for-loops in C or any other imperative language arise from elementary properties of the underlying maths domain.

Let us start by showing how the arithmetic operation of multiplying two natural numbers is a for-loop which emerges solely from the algebraic properties of multiplication:

$$\left\{ \begin{array}{l} a \times 0 = 0 \\ a \times 1 = a \\ a \times (b + c) = a \times b + a \times c \end{array} \right. \tag{1}$$

Just let $c := 1$ in the third (distributive property, obtaining $a \times (b+1)\ =\ a \times b + a \times 1$ and then simplify. The second clause is useful in this simplification

but it is not required in the final system of two equations,

$$\begin{cases} a \times 0 = 0 \\ a \times (b+1) = a \times b + a \end{cases} \tag{2}$$

since it is derivable from the two, for $b := 0$ and property $0 + a = a$ of addition. This system *is already* a runnable program in a functional language such as Haskell and many others.

The moral of this trivial exercise is that programs *arise* from the underlying maths, instead of being *invented* or coming out of the blue. Haskell 1st year students do this kind of reasoning all the time without noticing, in the first programs they write. For instance, the function which computes discrete exponentials will scale up this thanks to the properties

$$\begin{cases} a^0 = 1 \\ a^1 = a \\ a^{b+c} = a^b \times a^c \end{cases}$$

where the program just developed for multiplication can be re-used, and so on.

Type-wise, the multiplication algorithm just derived for natural numbers is not immediate to generalize. Intuitively, it will diverge for b a negative integer and for b a real number less than 1, at least. Argument a, however, does not seem to be constrained.

Indeed, the two arguments a and b will have different types in general. Let us see why and how. We start by looking at infix operators $\times$ and $+$ as curried operators — which is what they are in Haskell, cf. eg. type $(*) :: (Num\ a) \Rightarrow a \to a \to a$. Thus we can resort to the corresponding sections and write:

$$\begin{cases} (a\times)\ 0 = 0 \\ (a\times)\ (b+1) = (a+)\ ((a\times)\ b) \end{cases}$$

It can be easily checked that $(a\times) = \mathsf{for}\ (a+)\ 0$ where the **for-loop** combinator is given by

$$\begin{cases} \mathsf{for}\ f\ i\ 0 = i \\ \mathsf{for}\ f\ i\ (n+1) = f\ (\mathsf{for}\ f\ i\ n) \end{cases} \tag{3}$$

where f is the loop-body and i is the initialization value. In fact, for $f\ i\ n$ = $f^n i$, that is, f is iterated n times over the initial value i. For loops are a primitive construct available in many programming languages. In C, for instance, we will write something like

```
int mul(int a, int n)
{
int s=0; int i;
for (i=1;i<n+1;i++) {s += a;}
return s;
};
```

for (the uncurried version of) for-loop for $(a+)\ 0$.

To better understand this construct let us remove variables from both equations by lifting function application to function composition and lifting 0 to the "everywhere 0" function:

$$\begin{cases} (a\times)\cdot\underline{0} = \underline{0} \\ (a\times)\cdot(+1) = (+a)\cdot(a\times) \end{cases}$$

Using the *junc* ("either") pointfree combinator we merge the two equations into a single one,

$$[(a\times)\cdot\underline{0}\,,(a\times)\cdot(+1)] \quad = \quad [\underline{0}\,,(+a)\cdot(a\times)]$$

(thanks to the Eq-$+$ rule) single out the common factor $(a\times)$ on the left hand side,

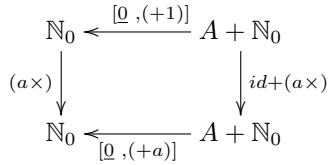
$$(a\times)\cdot[\underline{0}\,,(+1)] \quad = \quad [\underline{0}\,,(+a)\cdot(a\times)]$$

(thanks to $+$-fusion) and do a similar fission operation on the other side,

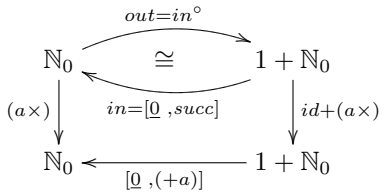
$$(a\times)\cdot[\underline{0}\,,(+1)] \quad = \quad [\underline{0}\,,(+a)]\cdot(id+(a\times)) \tag{4}$$

thanks to $+$-absorption.

Equalities of compositions are nicely drawn as diagrams. That of (4) is as follows:



Function $(+1)$ is the successor function $succ$. Type A is any (non-empty) type. For the particular case $A = 1$ the diagram is more interesting, as $[\underline{0}\,,succ]$ becomes an isomorphism, telling a unique way to build natural numbers: every natural number either is 0 or the successor of another natural number. Thus:

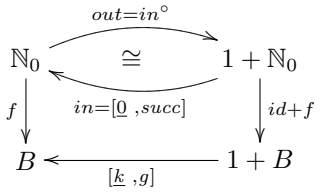


By solving the isomorphism equation $out\cdot\mathbf{in} = id$ we easily obtain the definition of out, the converse of $\mathbf{in}$ [1]:

[1] Note how the singularity of type 1 ensures out a function: what would be the outcome of $out\ 0$ in case A were arbitrary?

$$out\ 0 = i_1\ ()$$
$$out\ (n+1) = i_2\ n$$

Finally, we generalize $(+a)$ to any function $g :: B \to B$ and 0 to any constant k in B (thus B is assumed non-empty). The corresponding generalization of $(a\times)$ is denoted by f below:



It turns out that, given k and g, there is a unique solution to the equation (in f) captured by the diagram. We know this solution already, recall (3):

$$f \quad = \quad \mathsf{for}\ g\ k$$

Such uniqueness of solutions leads us to yet another universal property, that of for-loops:

$$f = \mathsf{for}\ g\ k \quad \equiv \quad f \cdot \mathbf{in} = [\underline{k}\ , g] \cdot (id + f) \tag{5}$$

From this property we infer the theory of for-loops in the standard way. For instance, making $f = id$ and solving the equation for g and k we obtain the for-reflexion law:

$$\mathsf{for}\ succ\ 0 \quad = \quad id \tag{6}$$

More knowledge about for-loops can be extracted from (5). Moreover, more elaborate and interesting for-loops arise from (5) and the law of mutual recursion. Come to lecture T-11 next week and read section 3.15 of [2].

Following a standard notation and terminology, and bearing a generalization to come up soon, we will refer to such a unique solution as *the catamorphism* [1] over the natural numbers induced by k and g and resort to the so-called "banana parentheses" to denote it: $(\!|[\underline{k}\ , g]|\!)$. This notation is more general and enables us to treat many different programming schemes as if they were abstractly the same. This will be addressed from lecture T-12 onwards.

References

[1] R. Bird and O. de Moor. *Algebra of Programming*. Series in Computer Science. Prentice-Hall International, 1997. C.A.R. Hoare, series editor.

[2] J.N. Oliveira. Program Design by Calculation, 2008. Draft of textbook in preparation (since 1998). Informatics Department, University of Minho. The following chapters are available from the author's website: *An introduction to pointfree programming*, *Recursion in the pointfree style*, *Why monads matter* and *Quasi-inductive datatypes*.

[3] David Lorge Parnas. Really rethinking "formal methods". *IEEE Computer*, 43(1):28–34, 2010.