From Curry to Haskell

Felice Cardone

Dipartimento di Informatica
Università di Torino

17 October 2018

Many thanks to Simone who is reading this while I am elsewhere.

A question

Where do the abstractions that we use in programming come from?

A question

Where do the abstractions that we use in programming come from?

- Model-View-Controller design pattern
- tacit (or point-free) programming
- "origami programming" (systematic use of fold functionals, say in Haskell)

A question

Where do the abstractions that we use in programming come from?

- Model-View-Controller design pattern
- tacit (or point-free) programming
- "origami programming" (systematic use of fold functionals, say in Haskell)

Many answers.

General answer

- The contributions of logic to the design of programming languages

General answer

- The contributions of logic to the design of programming languages
  - The contributions of logic, algebra and category theory to a programming style

General answer

- The contributions of logic to the design of programming languages
  - The contributions of logic, algebra and category theory to a programming style
    - The contributions of combinatory logic, algebra and category theory to a programming style

Leading example

Here is an example where all these aspects are present:

```
data Nat = Zero | Succ Nat

foldn ::  (a -> a) -> a -> Nat -> a
foldn h c Zero = c
foldn h c (Succ n) = h (foldn h c n)

add m n = foldn Succ m n
mult m n = foldn (+ m) Zero n
exp m n = foldn (* m) (Succ Zero) n
```

Leading example

Here is an example where all these aspects are present:

```
data Nat = Zero | Succ Nat
```
algebraic data type

```
foldn ::  (a -> a) -> a -> Nat -> a
```
fold functional

```
foldn h c Zero = c
foldn h c (Succ n) = h (foldn h c n)
```
applicative syntax

```
add m n = foldn Succ m n
mult m n = foldn (+ m) Zero n
exp m n = foldn (* m) (Succ Zero) n
```
universality of fold

Historical roots of this style of programming

Folding in APL

```
      ⍳5
1 2 3 4 5
      +/⍳5
15
```

Historical roots of this style of programming: Burstall 1969

Algebraic types

An α-*list* is either a *cons* or a *nil*.
A *cons* has an α and a *list*.
A *nil* has no components.

Structural recursion

let rec *concat*(*xs*1, *xs*2) = **cases** *xs*1:
x :: *xs*1: *x* :: *concat*(*xs*1, *xs*2)
nil( ) : *xs*2

Folding functional

let rec *lit*(*f*, *xs*, *y*) = **cases** *xs*:
x :: *xs*: *f*(*x*, *lit*(*f*, *xs*, *y*))
nil( ): *y*

Data types and formal systems

Curry's relevance to programming languages

Haskell Brooks Curry
(1920)

Curry's relevance to programming languages

Haskell Brooks Curry
(1920)

1927 Invention of combinators ($\mathbf{I}, \mathbf{W}, \mathbf{B}, \mathbf{C}$)

Curry's relevance to programming languages

Haskell Brooks Curry
(1920)

1927 Invention of combinators ($\mathbf{I}, \mathbf{W}, \mathbf{B}, \mathbf{C}$)

1934 Theory of functionality;
"propositions as types"

Curry's relevance to programming languages

Haskell Brooks Curry
(1920)

1927 Invention of combinators ($\mathbf{I}, \mathbf{W}, \mathbf{B}, \mathbf{C}$)

1934 Theory of functionality;
"propositions as types"

1939 Outlines of a Formalist Philosophy of Mathematics

Curry's relevance to programming languages

Haskell Brooks Curry
(1920)

1927 Invention of combinators ($\mathbf{I}, \mathbf{W}, \mathbf{B}, \mathbf{C}$)

1934 Theory of functionality;
"propositions as types"

1939 Outlines of a Formalist Philosophy of Mathematics

1958 (with R. Feys) Combinatory Logic I

Overview

In this talk I focus on neglected aspects of Curry's work:

- formal systems and syntax and their influence on programming,

Omitted in this talk:

- application as the only term-forming operation
- functional types

Overview

In this talk I focus on neglected aspects of Curry's work:

- formal systems and syntax and their influence on programming,
- the convergence of formal systems and algebraic notions as sources of ideas for abstractions in programming, in particular:
  - word algebras, and
  - their universal properties

Omitted in this talk:

- application as the only term-forming operation
- functional types

Special types of formal systems

A basic kind of formal system is the ob system, built from:

- Obs, formed from a set Ω of operations and atoms. Obs constructed by different processes are distinct.
- Elementary statements, formed by applying predicates (of degree n) to n obs.
- Axioms, which are statements, and rules for inferring statement from other statements, for example, for natural numbers:

$$\frac{}{0 \text{ is an } N}\ (\text{zero}) \qquad \frac{n \text{ is an } N}{s(n) \text{ is an } N}\ (\text{succ})$$

A formal system for lists of type A

- **Axiom**

$$\frac{}{Nil \text{ is a } List\ A}\ (\text{Nil})$$

- **Rule**

$$\frac{a \text{ is an } A \quad \ell \text{ is a } List\ A}{Cons(a, \ell) \text{ is a } List\ A}\ (\text{Cons})$$

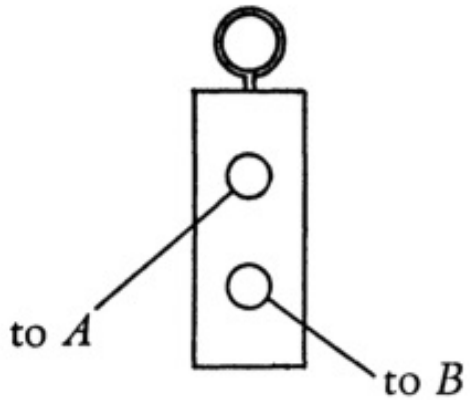
Formal systems as syntax

$$List\ A ::= Nil \mid Cons(A, List\ A)$$
$$A ::= a \mid \ldots$$

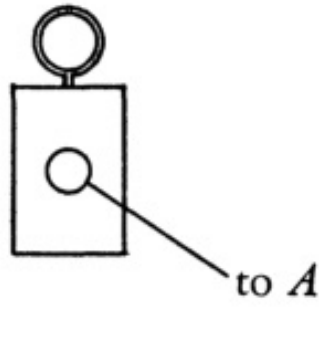
Abstract syntax (McCarthy, 1962)

- The Backus normal form that is used in the ALGOL report, describes the morphology of ALGOL programs in a synthetic manner. Namely, it describes how the various kinds of programs are built up from their parts. [Abstract syntax] is analytic rather than synthetic; it tells how to take a program apart, rather than how to put it together.
- Second, [abstract syntax] is abstract in that it is independent of the notation used to represent, say sums, but only affirms that they can be recognized and taken apart.

Example: an ob



$A \circ B$



A'

the holes are for the attachment of strings from the constituents A, B while the rings are for attachment of a string connecting to an operation when the term $A \circ B$ or A' is itself a constituent of a more complex term

(Curry, 1941)

Example: an ob

Alexander Calder, Mobile (c. 1932)

Tate Modern, London

Algebraic approach to abstract syntax

$$List\ A \underset{\text{constructors}}{\overset{\text{destructors}}{\rightleftarrows}} \{\underbrace{Nil}_{Nil}\} \cup (\underbrace{A \times List\ A}_{Cons})$$

- constructors synthesize lists, destructors analyze lists:

a List	is a	Cons
	or	Nil
a Cons	has an	A
	and a	List
Nil	has no	components

Algebraic approach to abstract syntax

$$List\ A \underset{\text{constructors}}{\overset{\text{destructors}}{\rightleftarrows}} \underbrace{\{Nil\}}_{Nil} \cup \underbrace{(A \times List\ A)}_{Cons}$$

- constructors synthesize lists, destructors analyze lists:

a List	is a	Cons
	or	Nil
a Cons	has an	A
	and a	List
Nil	has no	components

- *List A* fixed point of the functor $T(X) = \{Nil\} \cup (A \times X)$.

Algebraic approach to abstract syntax

$$List\ A \underset{\text{constructors}}{\overset{\text{destructors}}{\rightleftarrows}} \underbrace{\{Nil\}}_{Nil} \cup \underbrace{(A \times List\ A)}_{Cons}$$

- constructors synthesize lists, destructors analyze lists:

a List	is a	Cons
	or	Nil
a Cons	has an	A
	and a	List
Nil	has no	components

- $List\ A$ fixed point of the functor $T(X) = \{Nil\} \cup (A \times X)$.
- The initial T-algebra gives one such fixed point (Lambek Lemma)

Processes, products and free structures

Obs and their formation

In an ob system, an operation ω (of degree n) forms an ob b from n obs $a_1, \ldots, a_n$, denoted by

$$\omega(a_1, \ldots, a_n)$$

> *the genesis of such a b from $\omega, a_1, \ldots, a_n$ will also be referred to as a formation (of b), and for this formation ω will be called the operation, $a_1, \ldots, a_n$ the arguments, and b the closure* *(Curry, 1952)*

An aside: why is this interesting?

- Distinction between act and product (Twardowski, *Actions and Products*, 1912)
- Examples of the ambiguity act/product:

  construction, judgement, proof, organization,...

  Carl Adam Petri in his late unpublished work tried to cope with this ambiguity by means of his nets.
- The distinction has been refined for judgements by Sundholm and Martin-Löf to that between act, product, and content. For certain kinds of acts (e.g., proof) we also have their trace.

The anatomy of an ob

We distinguish:

- the formation of the ob $b = \omega(a_1, \ldots, a_n)$ (act), denoted by

$$\omega \cdot \langle a_1, \ldots, a_n \rangle$$

- the closure of a formation ξ (result) denoted by $\overline{\xi}$.

We define the ob $\omega(a_1, \ldots, a_n)$ like Curry did, as the closure of a formation from ω and arguments $a_1, \ldots, a_n$:

$$\omega(a_1, \ldots, a_n) \equiv \overline{\omega \cdot \langle a_1, \ldots, a_n \rangle}$$

The anatomy of an ob

Given a class of operations (a *signature*) Ω, define formal systems Ω^* and $\Omega(X)$ for any formal system X.

- Ω is the type of operations, Ω_n that of operations of degree n,
- $\Omega(X)$ is the type of *formations* with operation in Ω and arguments in X,
- Ω^* is the type of *obs* built from operations in Ω.

The anatomy of an ob

Let $a :: A$, when A is a formal system, mean a is an ob of A:

$$\frac{\omega \in \Omega_n \quad x_1 :: X, \ldots, x_n :: X}{\omega \cdot \langle x_1, \ldots, x_n \rangle :: \Omega(X)} \text{ (formation)}$$

$$\frac{\omega \cdot \langle a_1, \ldots, a_n \rangle :: \Omega(\Omega^*)}{\omega(a_1, \ldots, a_n) :: \Omega^*} \text{ (closure)}$$

Constructing an ob

For example, we get for $\Omega = \{\omega_2, \sigma_1, \gamma_0\}$:

$$\dfrac{\dfrac{\dfrac{}{\gamma_0 \cdot \langle\ \ \rangle :: \Omega(\Omega^*)}\ (\text{formation})}{\gamma_0 :: \Omega^*}\ (\text{closure}) \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\gamma_0 \cdot \langle\ \ \rangle :: \Omega(\Omega^*)}\ (\text{formation})}{\gamma_0 :: \Omega^*}\ (\text{closure})}{\sigma_1 \cdot \langle \gamma_0 \rangle :: \Omega(\Omega^*)}\ (\text{formation})}{\sigma_1(\gamma_0) :: \Omega^*}\ (\text{closure})}{\dfrac{\omega_2 \cdot \langle \gamma_0, \sigma_1(\gamma_0) \rangle :: \Omega(\Omega^*)}{\omega_2(\gamma_0, \sigma_1(\gamma_0)) :: \Omega^*}\ (\text{closure})}\ (\text{formation})$$

Effective transformations of formal systems

Given formal systems A and B we can define a notion of constructive transformation f from obs of A to obs of B:

$$f : A \longrightarrow B$$

means

f is an effective process that transforms $a :: A$ into $f(a) :: B$

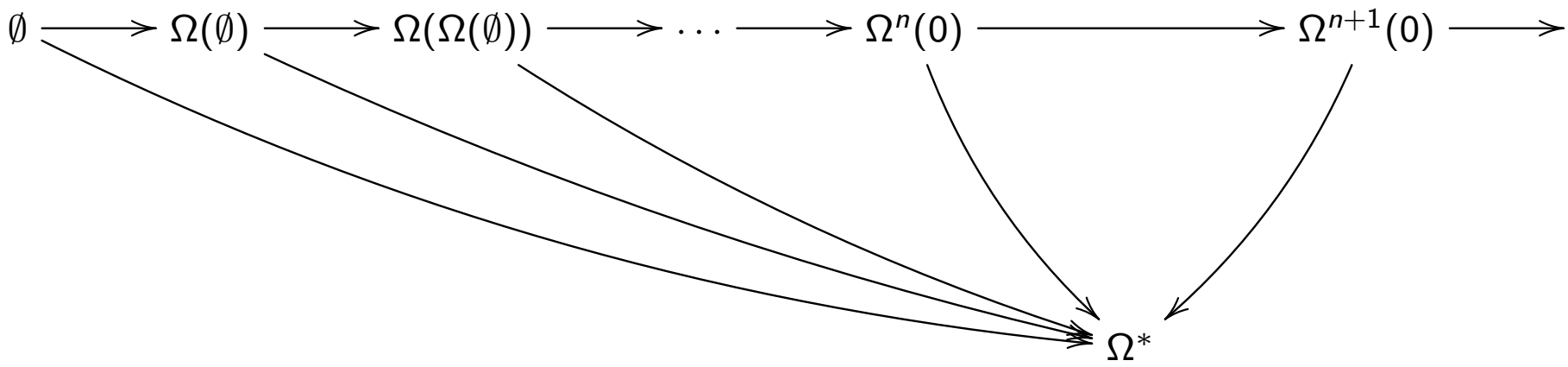
using a "definition" of effective processes given by Curry himself. So we get

$$\delta : \Omega(\Omega^*) \longrightarrow \Omega^*$$

and this is obtained by one application of rule (closure).

Constructing an ob

By iterating $\Omega(\cdot)$ we get a diagram whose spine represents the finite stages of the process of constructing obs, obtained by taking closures of the formations from the inside out by means of δ:



Formal systems and the standard algebraic approach

- Ω^* is "the smallest" closed under the process:

Formal systems and the standard algebraic approach

- Ω^* is "the smallest" closed under the process:
- $\delta : \Omega(\Omega^*) \longrightarrow \Omega^*$ corresponds to the initial Ω-algebra.

Formal systems and the standard algebraic approach

- Ω^* is "the smallest" closed under the process:
- $\delta : \Omega(\Omega^*) \longrightarrow \Omega^*$ corresponds to the initial Ω-algebra.
- For any other process

$$\xi : \Omega(X) \longrightarrow X$$

we can build an effective transformation $h : \Omega^* \longrightarrow X$.

Formal systems and the standard algebraic approach

- Ω^* is "the smallest" closed under the process:
- $\delta : \Omega(\Omega^*) \longrightarrow \Omega^*$ corresponds to the initial Ω-algebra.
- For any other process

$$\xi : \Omega(X) \longrightarrow X$$

  we can build an effective transformation $h : \Omega^* \longrightarrow X$.
- When $\Omega = \{0, s\}$, this gives the `foldn` functional.

Inversion and structural recursion

Lorenzen's calculi

Lorenzen (1955) studied calculi with rules of the form

$$\frac{A_1 \quad \ldots \quad A_n}{A}$$

where A_i, A are strings of atomic figures from a finite alphabet and variables

- *Atomic figures*: $+, o$
- *Variables*: X
- *Axiom*: $+$
- *Rules*

$$\frac{X}{Xo}\ (R1) \qquad \frac{X}{+X+}\ (R2)$$

Example:

$$\dfrac{\dfrac{\dfrac{\dfrac{+}{+o}\ (R1)}{+oo}\ (R1)}{++oo+}\ (R2)}{++oo+o}\ (R1)$$

Admissible rules

Admissibility

A rule R is admissible relative to a calculus K if any derivation obtained by the use of R can be constructively transformed into one obtained without such use (this definition of admissibility is due to Curry!)

Admissible rules

Admissibility

A rule R is admissible relative to a calculus K if any derivation obtained by the use of R can be constructively transformed into one obtained without such use (this definition of admissibility is due to Curry!)

Example The rule

$$\frac{X}{++X}$$

is admissible

Admissible rules

How could X have been derived using rules

$$\frac{X}{Xo}\ (R1) \qquad \frac{X}{+X+}\ (R2) \qquad ?$$

For example

$$\frac{\vdots}{\underbrace{X'o}_{X}}\ (R1) \qquad \text{is transformed into} \qquad \frac{\begin{array}{c}\vdots\\ X'\end{array}}{+\underbrace{+X'o}_{X}}\ (R1)$$

Inversion principle

Inversion principle

Elimination rules follow from introduction rules (Gentzen)

Inversion principle

Elimination rules follow from introduction rules (Gentzen)

If, in a calculus K, A can only be introduced by the rules

$$\frac{\Gamma_1}{A} \qquad \frac{\Gamma_n}{A}$$

then the elimination rule

$$\frac{A}{C}$$

is admissible relative to K when

$$\frac{\Gamma_1}{C} \qquad \frac{\Gamma_n}{C}$$

are used as additional rules

Inversion and structural recursion

Natural numbers N

There are two introduction rules for N

$$\frac{}{0 :: N} \qquad \frac{n :: N}{s(n) :: N}$$

An "elimination rule"

$$\frac{n :: N}{a :: A}$$

is admissible when there is an effective process f that transforms any ob of N into an ob of A, that is when

$$f : N \longrightarrow A$$

`foldn` as an elimination rule

Assume we have an ob $a :: A$ and a process $f : A \longrightarrow A$.

We can define the elimination process

$$\texttt{foldn}\ f\ a : N \longrightarrow A$$

by induction on the height of the derivation of the judgement $n :: N$:

$$\begin{aligned} \texttt{foldn}\ f\ a\ 0 &= a && \text{if } n = 0 \\ \texttt{foldn}\ f\ a\ (s(x)) &= f(\texttt{foldn}\ f\ a\ x) && \text{if } n = s(x) \end{aligned}$$

Definition by recursion (Dedekind, 1888)

Natural numbers as iterators

Natural numbers are iterators

- Wittgenstein's Tractatus
- Church numerals in λ-calculus
- numbers as arguments of `foldn` f a
  (Martin-Löf and Hancock, 1975)

The revenge of concrete syntax

How to prove the existence of initial algebras?

Some well-known facts:

The revenge of concrete syntax

How to prove the existence of initial algebras?

Some well-known facts:

- The elements of Ω^* admit a linear representation without parentheses (Łukasiewicz, 1929), in particular one in which every operation symbol follows its arguments (postfix polish notation)

The revenge of concrete syntax

How to prove the existence of initial algebras?

Some well-known facts:

- The elements of Ω^* admit a linear representation without parentheses (Łukasiewicz, 1929), in particular one in which every operation symbol follows its arguments (postfix polish notation)
- There is an iterative algorithm for parsing obs written in postfix polish notation (Menger, 1930; Schröter, 1943; Gerneth, 1948; Hall, c. 1950; Rosenbloom, 1950):
  1. let the rank of an atom be 1, the rank of an operation of degree n be $-n+1$.
  2. scan from left to right the sequence of symbols adding their ranks: if the partial sums are never negative and final sum is 1, the sequence represents an ob

The revenge of concrete syntax

How to prove the existence of initial algebras?

Some well-known facts:

- The elements of Ω^* admit a linear representation without parentheses (Łukasiewicz, 1929), in particular one in which every operation symbol follows its arguments (postfix polish notation)
- There is an iterative algorithm for parsing obs written in postfix polish notation (Menger, 1930; Schröter, 1943; Gerneth, 1948; Hall, c. 1950; Rosenbloom, 1950):
  1. let the rank of an atom be 1, the rank of an operation of degree n be $-n+1$.
  2. scan from left to right the sequence of symbols adding their ranks: if the partial sums are never negative and final sum is 1, the sequence represents an ob
- There is an algorithm for evaluating an ob represented in postfix Polish (Bauer and Samelson; Burks, Warren and Wright, 1954)

Conclusions

A question – take 2

Where do the abstractions that we use in programming come from?

A question – take 2

Where do the abstractions that we use in programming come from?

More answers:

A question – take 2

Where do the abstractions that we use in programming come from?

More answers:

- ideas from the computer as an information machine
  $\rightsquigarrow$ **Engelbart's Thesis**

A question – take 2

Where do the abstractions that we use in programming come from?

More answers:

- ideas from the computer as an information machine
  $\rightsquigarrow$ **Engelbart's Thesis**
- ideas from early semiotic studies
  $\rightsquigarrow$ **Carnap, Curry, Gorn, Caracciolo, Zemanek,...**

Thanks for your presence during my absence.