The mathematical language, the programming languages, etc.

Gilles Dowek

École polytechnique

From scores to programs

Changing the scale

$$(3/2)^{53}/2^{31} = 1.0020$$

much better than $(3/2)^{12}/2^{7} = 1.0136$

A scale with 53 intervals

- makes it more difficult to design an instrument
- makes it more difficult to play the instrument
- requires a new way to write music

n the XIX^e century almost impossible

With electronic instruments : difficult, but not impossible

Using tapes as instruments (phasing)

Inspired by) *Clapping music* (Reich, 1972).

A one minute piece. One musician claps 61 times (every second)
Another 62 times (every 60/61) seconds

Difficult to play without a device (but easy with tapes or
electronic instruments)

Only way to write down this piece with traditional notation:

et the tempo to $60 \times 61 = 3660$ quarter-note/mn

Then in one score between two notes 61 rests and 60 for the other

Not very useful

A compact notation

```
for i = 0 to  3660
do if i mod 61 = 0 then clap () done
```

The notion of complexity (in the tradition of Kolmogorov)

Size of the shortest program generating some datum

The sequence `1 2 3 4 5 6 7 ...` has a simpler complexity that `4 6 2 6 3 2 9 ...`

Maximal complexity : random sequence (shortest program = sequence)

Minimal complexity : Bach's fugues

A real time issue

Two scores :

```
for i =  0 to  3660
do if i mod 61 = 0 then clap () done
```

and

```
for i =  0 to  3660
do if i mod 60 = 0 then clap () done
```

- a rest needed when `i` is not a multiple of `60`
- some kind of synchronization of the musicians is needed
- the tempo has to be taken into account

One step back

A score expresses some object

That can be expressed in other languages (e.g. programming anguage)

But what kind of object does a score express ?

A function of time (continuous variable)

or a sequence (function of a discrete variable)

Traditional notation: a score is a function of a discrete variable

Lists

Discrete variable, finite time: finite sequence

Finite sequences are not functions but lists

```
let c n = if n mod 61 = 0 then Clap else Rest
in build 0 3660 c
```

```
[Clap; Rest; Rest; ...  Rest; Clap; Rest; ...]
```

The program is a better description that the list

- shorter
- the intention of the composer are easier to understand

Streams (infinite lists)

The finiteness may be a key issue or not

```
let s =
let c n = if n mod 61 = 0 then Some Clap else Some Rest
in from c
```

```
note Stream.t = <abstr>
```

An infinite list `s` is “built”

Each single individual element has not been computed yet

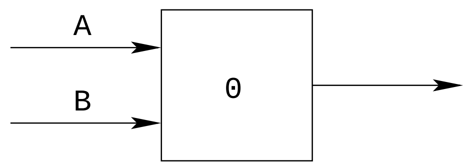
Computed on demand

`next s` : `Clap`

`next s` : `Rest`

Stream languages

- Stream languages : *Synchronous languages* (*Lustre, Lucid synchone, Signal, Esterel, ...*)
- Reactive systems



- In state `0` (resp. `1`) transmit `A` (resp. `B`) and switch to `1` (resp. `0`)
- From the streams `3 1 4 1 5 9 2 6 5 3 5 ...` and `0 1 2 3 4 5 6 7 8 9 0 ...` builds `3 1 4 3 5 5 2 7 5 9 5`
- Other examples : add one fifth, delay, ...
- Catch

From programs to specifications

Another step back

nstead of writing one note after the other

we can describe the score in a more abstract way

as a program building an object (function, list, stream, ...)

Computational ?

Programs are computational definitions of objects

computational : music must be performed, after all

Computability can come in a second step

There is a time to define, and a time to compute

How do we define a function in mathematic ?

We define a relation

$$x = 2y \text{ or } x = 2y + 1$$

$(0, 0), (1, 0), (2, 1), (3, 1), (4, 2), (5, 2), ...$

Then we prove that this relation is functional

$$\forall x\ \exists_1 y\ (x = 2y \text{ or } x = 2y + 1)$$

The relation $x = 2y$ or $x = 2y + 1$ tells what to compute, not how

Computable *a posteriori*

The function

$$x \mapsto [y \mid x = 2y \text{ or } x = 2y + 1]$$

is computable

But it is not defined as an algorithm

Underspecification

Sometimes the relation is not functional

$$2x \leq y$$

We cannot prove

$$\forall x \; \exists_1 y \; 2x \leq y$$

yet we can prove

$$\forall x \; \exists y \; 2x \leq y$$

The relations defined a familly of function $x \mapsto 2x + 5$, $x \mapsto 3x$, ...

Some of which may be computable

Proofs of programs

The relation

$$x = 2y \text{ or } x = 2y + 1$$

is a specification

Define the function another time as an algorithm

```
let rec div x = if x <= 1 then 0 else 1 + div(x - 2)
```

Prove

$$y = f(x) \Leftrightarrow R(x, y)$$

(or $y = f(x) \Rightarrow R(x, y)$ if underspecified)

More specifications

n this case, any property of the function can serve as a specification

$$\forall t \ f(t + 60) = f(t)$$

$$\forall t \ g(t + 61) = g(t)$$

pecifies *Clapping music* and many other phasing pieces

Specifying data

When the specified object is a function

$$\forall x \exists y \; R(x, y)$$

When the specified object is a datum (e.g. list)

$$\exists y \; P(y)$$

e.g. $\exists y \forall n \; (nth \; n \; y) = (nth \; (n + 61) \; y)$

Can we specify non computable functions ?

Yes:

$$(y = 1 \text{ and } (x \text{ terminates})) \text{ or } (y = 0 \text{ and } \text{not}(x \text{ terminates}))$$

Defines the function that maps a program to 1 if the program
erminates and 0 otherwise

This function is not computable (Turing, Church-Kleene, 1936)

Can we define non computable functions ?

But can we prove

$$\forall x\ \exists y\ ((y = 1 \text{ and } (x \text{ terminates})) \text{ or } (y = 0 \text{ and not}(x \text{ terminates})))$$

?

Yes. The proof uses the fact that

$$(x \text{ terminates}) \text{ or not}(x \text{ terminates})$$

$$A \text{ or not } A$$

The excluded middle

A function whose proof of existence does not use the excluded middle (constructive proof) is always computable

The witness property

From a constructive proof of

$$\forall x \; \exists y \; S(x,y)$$

build a constructive proof of

$$\exists y \; S(n,y)$$

Then, from a constructive proof of a statement

$$\exists y \; S(n,y)$$

compute a witness, i.e. a datum p that verifies the property $S(n,p)$

For data specification

Even easier :

From a constructive proof of a statement

$$\exists y \; P(y)$$

compute a witness, i.e. a datum l that verifies the property $P(l)$

Programming with proofs *v.s.* constraint programming

$$S(x, y)$$

Programming with proofs: We build a constructive proof (either automatically or not) of $\forall x \exists y \; S(x, y)$

When we have an input n, we get a proof of $\exists y \; S(n, y)$ and we compute the witness from this proof (output)

Constraint programming: When we have an input n, we automatically) build a constructive proof of $\exists y \; S(n, y)$ and compute the witness from this proof (output)

Programming with specifications

Old fashion design: I have something in mind, I write it down step by step

A higher-level approach: (proofs of programs, programming with proofs, constraint programming)

The important part is the specification what, not the program how

Describe your expectation, let the computer fulfill it

If your expectation is underspecified one solution or another may be given to you : you accept to have only a partial control