Unified Media Programming: An Algebraic Approach

Simon Archipoff, **David Janin**,
LaBRI,
Bordeaux INP, University of Bordeaux

FARM@ICFP, Oxford, 9/9/2017

Plan of the talk

- Preamble: a glimpse of the future
- Opening: the turtle and its pen
- Theme I: monoid semantics
- Theme III: resettable monoid semantics
- Variation: the turtle and its time machine
- Finale: demo and conclusion

- More in the paper:Theme II: inverse monoid semantics

A glimpse of the futur

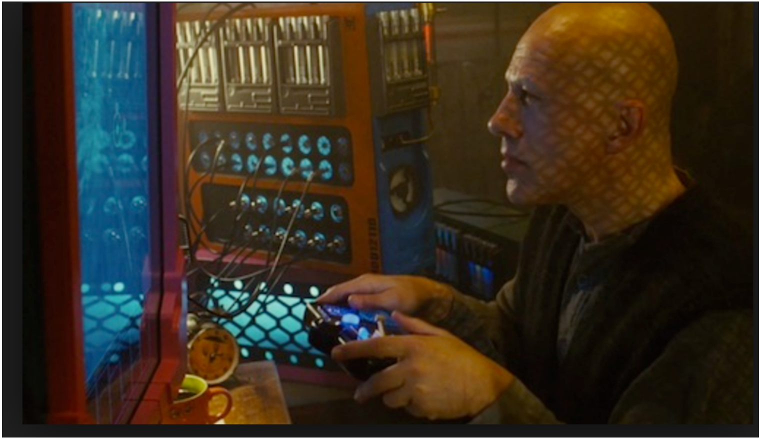
Preamble

Programer office

from *Zero Theorem* by Terry Gilliam, 2013.

Preamble

Programming device



from *Zero Theorem* by Terry Gilliam, 2013.

Preamble

Programming interface

from *Zero Theorem* by Terry Gilliam, 2013.

The Terry Gilliam “correspondance”’

$$Program = proof = buidling$$

and

$$Programmer = prover = builder$$

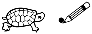
The turtle and its pen

Opening : a turtle equipped with a pen

Opening : the pen can be moved on or off the screen

Opening : the pen can be moved on or off the screen

Opening : the pen can be moved on or off the screen



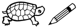
Opening : the pen can be moved on or off the screen

drawing a point when pen is on...

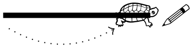
Opening : the turtle can walk

Opening : the turtle can walk

Opening : the turtle can walk



Opening : the turtle can walk



drawing segments when pen is on...

Opening : the turtle can walk

Opening : the turtle can walk

Opening : the turtle can walk

Opening : the turtle can walk

Opening : the turtle can walk

Opening : the turtle can walk

Opening : the turtle can walk

Opening : the turtle can walk

Opening : the turtle can walk

Opening : the turtle can turn

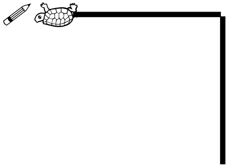
Opening : the turtle can turn

Opening : the turtle can turn

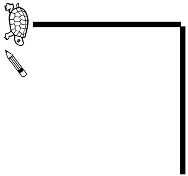
Opening : the turtle can turn

Opening : the turtle can turn

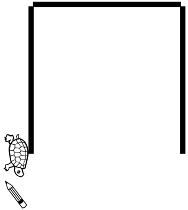
Opening : the turtle can turn



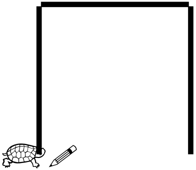
Opening : the turtle can turn



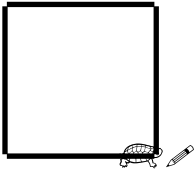
Opening : the turtle can turn



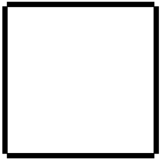
Opening : the turtle can turn



Opening : the turtle can turn



Opening : the turtle can turn



drawing 2D figures (or more)...

Program syntax

A program is a sequence of elementary actions

- *Flip* pen on or off
- *Walk d* for some distance d
- *Turn d* for some angle d

Program syntax

A program is a sequence of elementary actions

- *Flip* pen on or off
- *Walk* d for some distance d
- *Turn* d for some angle d

The square example

Walk 1, Turn $\pi/2$, Walk 1, Turn $\pi/2$, Walk 1, Turn $\pi/2$, Walk 1, Turn $\pi/2$!

Monoid semantics

Goal

Programs form a monoid. We aim at defining a monoid semantics model for turtle program.

Semantics elements : positions, figures, moves and drawings

State space

- Positions: $P = \mathbb{R}^2 \times \mathbb{R}/2\pi\mathbb{R} \times \mathbb{B}$,
- Figures: $F = \mathcal{P}(\mathbb{R}^2 \times \mathbb{R}^2)$.

Semantics elements : positions, figures, moves and drawings

State space

- Positions: $P = \mathbb{R}^2 \times \mathbb{R}/2\pi\mathbb{R} \times \mathbb{B}$,
- Figures: $F = \mathcal{P}(\mathbb{R}^2 \times \mathbb{R}^2)$.

Moves and drawings

- Moves: $M = P \to P$,
- Drawings: $D = P \to F$.

Monoid semantics

Elementary action semantics

Move semantics: $P \to P$

- $[\![\, Flip \,]\!]_M(p, a, b) = (p, a, \neg d)$
- $[\![\, Walk \ d \,]\!]_M(p, a, b) = (p + (d * cos(a), d * sin(a)), a, d)$
- $[\![\, Turn \ d \,]\!]_M(p, a, b) = (p, a + d, b)$

Monoid semantics

Elementary action semantics

Move semantics: $P \rightarrow P$

- $[\![\, Flip \,]\!]_M(p, a, b) = (p, a, \neg d)$
- $[\![\, Walk\ d \,]\!]_M(p, a, b) = (p + (d * cos(a), d * sin(a)), a, d)$
- $[\![\, Turn\ d \,]\!]_M(p, a, b) = (p, a + d, b)$

Drawing semantics: $P \rightarrow F$

- $[\![\, Flip \,]\!]_D(p, a, b) = \begin{cases} \{(p, p)\} \text{ when } b = 0 \\ \emptyset \text{ when } b = 1 \end{cases}$
- $[\![\, Walk\ d \,]\!]_D(p, a, b) = \begin{cases} \{(p, p + (d * cos(a), d * sin(a)))\} & \\ & \text{when } b = 1 \\ \emptyset & \text{when } b = 0 \end{cases}$
- $[\![\, Turn\ d \,]\!]_D(p, a, b) = \emptyset$

Monoid semantics

The monoid

Set $S = \underbrace{P \to P}_{M} \times \underbrace{P \to \mathcal{P}(F)}_{D}$ equipped with product

$$(m_1, d_1) \cdot (m_2, d_2) = (m_2 \circ m_1, d_1 \cup d_2 \circ m_1).$$

Monoid semantics

The monoid

Set $S = \underbrace{P \to P}_{M} \times \underbrace{P \to \mathcal{P}(F)}_{D}$ equipped with product

$$(m_1, d_1) \cdot (m_2, d_2) = (m_2 \circ m_1, d_1 \cup d_2 \circ m_1).$$

Program semantics

Inductively define by composition of elementary action semantics

- $[\![\epsilon]\!] = \emptyset$ with empty program ϵ,
- $[\![ap]\!] = ([\![a]\!]_M, [\![a]\!]_D) \cdot [\![p]\!]$ for all elementary action a and program p.

Monoid semantics

The monoid

Set $S = \underbrace{P \to P}_{M} \times \underbrace{P \to \mathcal{P}(F)}_{D}$ equipped with product

$$(m_1, d_1) \cdot (m_2, d_2) = (m_2 \circ m_1, d_1 \cup d_2 \circ m_1).$$

Program semantics

Inductively define by composition of elementary action semantics

- $[\![\, \epsilon \,]\!] = \emptyset$ with empty program ϵ,
- $[\![\, ap \,]\!] = ([\![\, a \,]\!]_M, [\![\, a \,]\!]_D) \cdot [\![\, p \,]\!]$ for all elementary action a and program p.

Remark

Semantics is the morphism generated from the (free) monoids of programs into the semantics monoids.

Monoid semantics

Lemma

- moves under (flipped) composition form a monoid,
- drawings under (element-wise) union form a monoid,
- moves act by endomorphisms over drawings by $m * d = d \circ m$,
- and we have $S = M \ltimes D$, i.e. semantics monoid is a semi-direct product.

Resettable monoid semantics

Resettable monoid semantics

Program reset

For every program p, define $reset(p)$ by $[\![\, reset(p) \,]\!] = (id, [\![\, p \,]\!]_D)$.

Resettable monoid semantics

Program reset

For every program p, define $\mathit{reset}(p)$ by $[\![\, \mathit{reset}(p) \,]\!] = (\mathit{id}, [\![\, p \,]\!]_D)$.

General construct

Given a monoid M acting by endomorphisms on a lattice L, the semi-direct product $M \ltimes L$ is a resettable monoid.

Resettable monoid semantics

Program reset

For every program p, define $reset(p)$ by $[\![\, reset(p) \,]\!] = (id, [\![\, p \,]\!]_D)$.

General construct

Given a monoid M acting by endomorphisms on a lattice L, the semi-direct product $M \ltimes L$ is a resettable monoid.
That is, given $(m, d)^R = (1, d)$, we have:

- $(M \ltimes L)^R = \{1\} \ltimes L$ is a idempotent commutative submonoid,
- $(m, d)^R$ is the least left unit of (m, d).

Resettable monoid semantics

Program reset

For every program p, define $reset(p)$ by $[\![\, reset(p) \,]\!] = (id, [\![\, p \,]\!]_D)$.

General construct

Given a monoid M acting by endomorphisms on a lattice L, the semi-direct product $M \ltimes L$ is a resettable monoid.
That is, given $(m, d)^R = (1, d)$, we have:

- $(M \ltimes L)^R = \{1\} \ltimes L$ is a idempotent commutative submonoid,
- $(m, d)^R$ is the least left unit of (m, d).

This is known in the York school as a *left semi-adequate* monoid.

Resettable monoid semantics

Program reset

For every program p, define $reset(p)$ by $[\![\, reset(p) \,]\!] = (id, [\![\, p \,]\!]_D)$.

General construct

Given a monoid M acting by endomorphisms on a lattice L, the semi-direct product $M \ltimes L$ is a resettable monoid.
That is, given $(m, d)^R = (1, d)$, we have:

- $(M \ltimes L)^R = \{1\} \ltimes L$ is a idempotent commutative submonoid,
- $(m, d)^R$ is the least left unit of (m, d).

This is known in the York school as a *left semi-adequate* monoid.
Moreover, for all $x, y, z \in M \ltimes L$, we have:

- $x^R = y^R \Rightarrow (zx)^R = (zy)^R$ (*left Ehresmann*),
- $(xy)^R x = xy^R$ (*left restriction*).

Resettable monoid semantics

Extending Turtle Programs

- Moves can be extended to non injective moves, e.g. projection.

Resettable monoid semantics

Extending Turtle Programs

- Moves can be extended to non injective moves, e.g. projection.

Interpretation of the reset

- Resets act as kind of a fork operator: $\mathit{reset}(p) <> q$ can be understood as "fork a sub turtle behaving like p" and "keep on executing q".

The turtle and its violin

The violin metaphor

We want our turtle “to play violin” or, more seriously, to act also over the time dimension...

Temporal turtle semantics

Temporal moves and animations

Let T be a timescale, e.g. $T = \mathbb{R}$.

- Temporal moves: $TM = T \to T$ (possibly partial),
- Animation $A = T \to S$,

where $S = (M, D)$ is the (inverse monoid of) turtle 2D semantics.

Temporal turtle semantics

Temporal moves and animations

Let T be a timescale, e.g. $T = \mathbb{R}$.

- Temporal moves: $TM = T \to T$ (possibly partial),
- Animation $A = T \to S$,

where $S = (M, D)$ is the (inverse monoid of) turtle 2D semantics.

Temporal semantics

- TM is a monoid under flipped composition,
- A is a monoid under point-wise extension of the 2D product,
- TM acts by endomorphism on A by $tm * a = a \circ tm$ (with $a \circ tm(t) = \emptyset$ in the case $tm(t)$ is undefined),

so that $TS = TM \ltimes A$ is a inverse (or at least resettable) monoid for temporal turtle program semantics.

Temporal turtle elementary programs

Examples

Bijection over time:

- $[\![\mathit{Delay}\ d]\!] = (t \mapsto t + d, \epsilon)$,
- $[\![\mathit{Strech}\ f]\!] = (t \mapsto t * f, \epsilon)$,

or partial bijection over time (extending action of TM over A accordingly)

- $[\![\mathit{Start}]\!] = (t \mapsto t \text{ if } t \geq 0, \epsilon)$,
- $[\![\mathit{Stop}]\!] = (t \mapsto t \text{ if } t \leq 0, \epsilon)$,

which can be combined with delay to define *Play* $t_1\ t_2$ that cuts the timescale from t_1 to t_2...

Temporal turtle elementary programs

Examples

Bijection over time:

- $[\![\, Delay\ d]\!] = (t \mapsto t + d, \epsilon)$,
- $[\![\, Strech\ f]\!] = (t \mapsto t * f, \epsilon)$,

or partial bijection over time (extending action of TM over A accordingly)

- $[\![\, Start]\!] = (t \mapsto t \text{ if } t \geq 0, \epsilon)$,
- $[\![\, Stop]\!] = (t \mapsto t \text{ if } t \leq 0, \epsilon)$,

which can be combined with delay to define *Play* $t_1\ t_2$ that cuts the timescale from t_1 to t_2...

A programing API

In a modern language such as Haskell, the above functions may be part of a *timed monoid* class type.

Musical extension

Let E be a set of musical events, e.g. note on n^+ and off n^- one for each note $n \in N$. Let T be the symbolic temporal scale.

Musical extension

Let E be a set of musical events, e.g. note on n^+ and off n^- one for each note $n \in N$. Let T be the symbolic temporal scale.

Symbolic music monoid

Consider $D = T \to T$ (partial) and $P = T \to \mathcal{P}(E \times T)$ with

$$M = D \ltimes P$$

and the induced (resettable) submonoid generated by

$$[\![\, n \,]\!] = (t \mapsto t + 1, t \mapsto \{(n^+, t), (n^-, t + 1)\})$$

one for each note $n \in N$ together with *Delay*, *Shift*, *Start* and *Stop*.

Musical extension

Let E be a set of musical events, e.g. note on n^+ and off n^- one for each note $n \in N$. Let T be the symbolic temporal scale.

Symbolic music monoid

Consider $D = T \to T$ (partial) and $P = T \to \mathcal{P}(E \times T)$ with

$$M = D \ltimes P$$

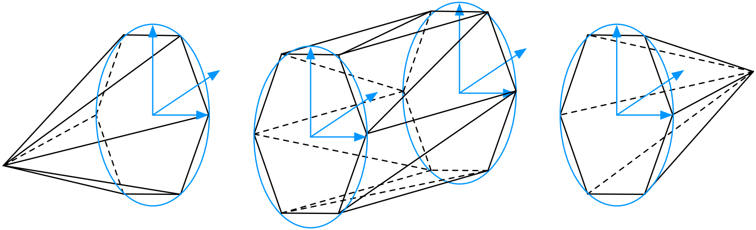
and the induced (resettable) submonoid generated by

$$[\![n]\!] = (t \mapsto t + 1, t \mapsto \{(n^+, t), (n^-, t + 1)\})$$

one for each note $n \in N$ together with *Delay*, *Shift*, *Start* and *Stop*. Then, the monoids M contains all finite polyphonic symbolic melodies.

Our actual implementation

3D by extrusion



Painting vs extruding

Extrusion is easier

- pen down : “control” points (or curves),
- move : combining complex moves,
- pen down : “control” points (or curves),
- move : combining complex moves,
- etc..

and compute *automatically*...curves (surfaces), curve tangents (or surface normals), resolutions, interpolations...

Painting vs extruding

Extrusion is easier

- pen down : “control” points (or curves),
- move : combining complex moves,
- pen down : “control” points (or curves),
- move : combining complex moves,
- etc..

and compute *automatically*...curves (surfaces), curve tangents (or surface normals), resolutions, interpolations...

Drawing specification

With reset, a drawing specification is a tree structures set of “control points”...

From Haskell to GPU

A three layer architecture

- Haskell animated 3D scene syntax,
- Haskell animated 3D scene specification,
- GPU rendering.

From Haskell to GPU

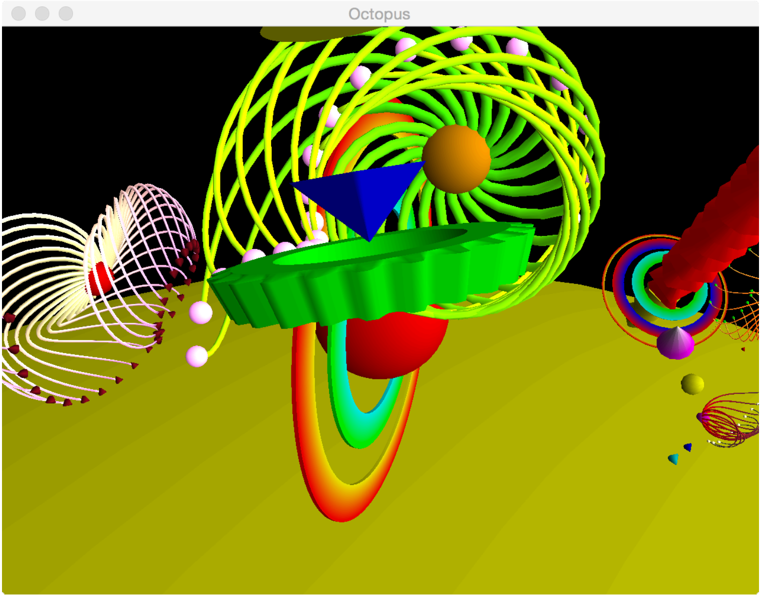
A three layer architecture

- Haskell animated 3D scene syntax,
- Haskell animated 3D scene specification,
- GPU rendering.

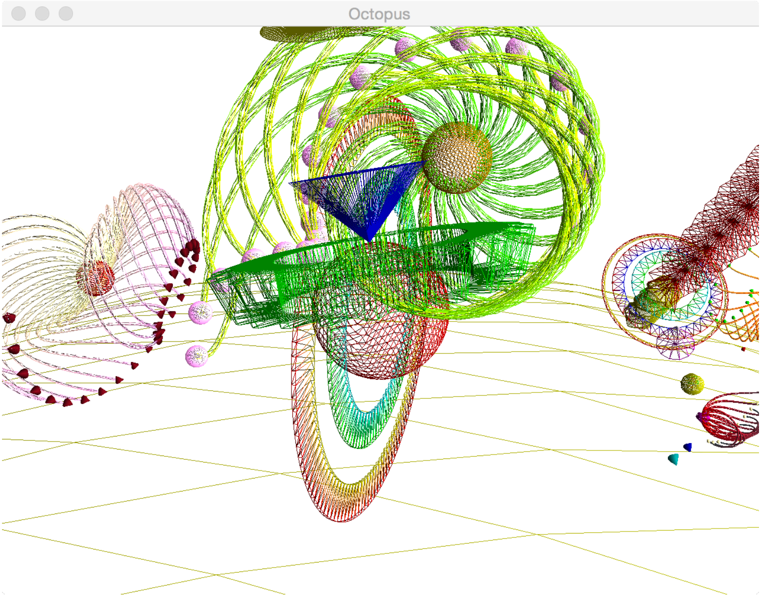
Efficiency

- 5000 drawing elements specified (on CPU)
- up to 1000000 triangles drawn (on GPU)
- at decent animation rate (from 30 to 70 fps).

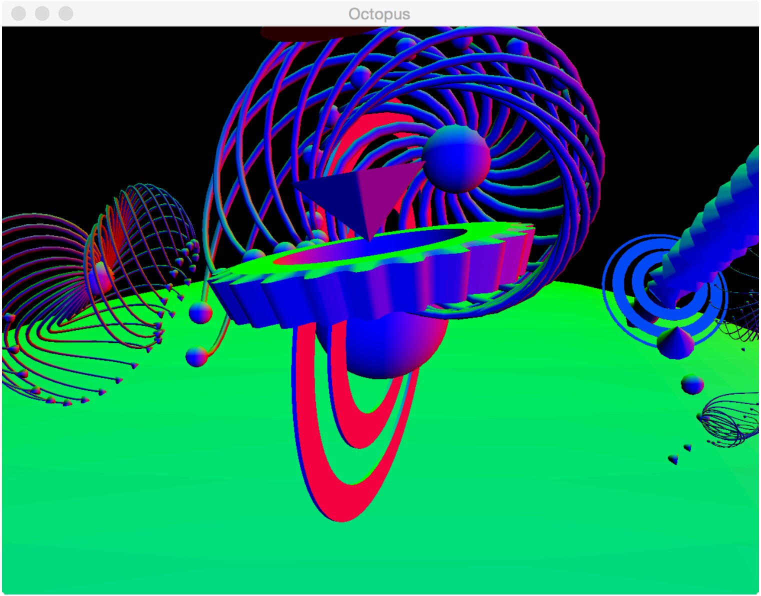
Drawing examples



Drawing examples



Drawing examples



To do list

Normal forms

The need for normal forms

So far, we have seen ways of *synthesizing/writing* temporal media. Temporal media transformations require ways of *analyzing/reading* temporal media. Normal forms should allow this.

Stronger needs when on-the-fly

Temporal media must be read (in reactive program) in a time coherent way.

Other needs

- Generic handling of piles of semi-direct product (much like Monad transformers)
- Cut monoids vs enveloppe monoids (dual construction ?)
- Richer geometry (with bounded size instances to be sent to the GPU),
- Heterogeneous dimension (points, curves, surfaces, volumes, etc...),
- Examples library (pandemonium)

Octopus on the grid:

https://github.com/OctopusFarm/Octopus