Unified Media Programming: An Algebraic Approach

Simon Archipoff
UMR CNRS LaBRI, University of Bordeaux
France
Simon.Archipoff@labri.fr

David Janin
UMR CNRS LaBRI, Bordeaux INP, University of Bordeaux
France
David.Janin@labri.fr

Abstract

In this paper, we aim at defining a simple and sound mathematical framework for describing temporal media programming language semantics. It occurs that semigroup theory offers various concepts that are especially well suited for this purpose.

As a result, a fairly general programming scheme can be defined in order to specify, compose and render both spatial media objects (e.g. 3D drawings) and timed media objects (e.g. Animation or Music). Each of these constructs is specified in Haskell via an adequate type class definition and an associated uniform data type construct.

A simple monoid based semantics model of the turtle command language of Logo is detailed and extended throughout the paper. This allows for providing step by step introductions and usage examples of the algebraic concepts and constructs our proposal is based on.

CCS Concepts • **Software and its engineering → Semantics**; **Domain specific languages**; • **Applied computing** → *Media arts*;

Keywords temporal animation, 3D programming, model based programming language

ACM Reference Format:
Simon Archipoff and David Janin. 2017. Unified Media Programming: An Algebraic Approach. In *Proceedings of 5th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling and Design, Oxford, UK, September 9, 2017 (FARM'17),* 12 pages.
https://doi.org/10.1145/3122938.3122943

1 Introduction

In this paper, we develop a semantics model for the synthesis of temporal media as defined by Hudak [7] such as animations, as illustrated in this paper, but also applicable to music, as illustrated in former work [2] or in Euterpea [8].

Spatio-temporal modeling. Focusing our attention on the synthesis of temporal media, programs can handle the time dimension quite in the same way drawing programs handle space dimension. It is only at the rendering stage, that is, when executing a program, that the time dimension specifically differs from space dimensions.

More precisely, when executing a temporal media program, in other words, when rendering a defined temporal media, time flow is irreversible, i.e. a figure that has been displayed on the screen cannot be "undisplayed". However, at the programming stage, that is, when specifying a temporal media program, nothing prevents a programmer/designer from going back and forth along the time dimension. A temporal media program is out-of-time while its execution is in-time [10]. This means that, when defining temporal media, both space and time dimensions can be treated similarly.

It follows that any programming construct that allows for drawing a picture, say on the two dimensional space defined by the screen, can also be used for defining the transformation of that picture along the time dimension when defining an animation. This simple observation and a careful study of the underlying semantics of a typical drawing programming language: Logo's turtle command, leads us to the definition of a kind of unified model for temporal media synthesis programming.

Semantics driven DSL definition. Aiming at defining a domain specific language (DSL) for temporal media programming, our approach is governed by semantics.

More precisely, we develop a semantic model for temporal media from which syntax can be derived. Our syntax will essentially reflect the way temporal media can be transformed and combined. It follows that, at the present stage, it remains mostly implicit. What does matter throughout our presentation is semantics.

Then one may ask what is the mathematical framework we use for our semantic model. Our answer is, somehow paradoxically, Haskell programs, that is, pieces of Haskell syntax! There is (essentially) no need for distinguishing program's syntax from program's semantics: every (pure) Haskell function (implicitly) denotes a unique (mathematical) function.

Incidentally, purity also forces all semantical dependencies to be made explicit; there is no possible "magic semantic trick" hidden in a side effect. However, despite such a complete and explicit semantic model definition, a simple application programming interface (API) can still be achieved. All the semantical dependencies that can uniformly (and automatically) be realized can be hidden into higher order programming constructs: mostly monoids, but also monads.

The program constructs we propose throughout this paper are essentially based on combining, by means of a single uniform associative binary operator, various pieces of programs that both act on space (or time) dimensions and produce media values.

Main contribution. We first show, as announced above, that spatial and temporal dimensions semantics can indeed be treated (mostly) in the same way. For this purpose, we develop an original algebraic semantics for temporal media programming languages.

Our semantic model is based on semigroup theoretical concepts such as semigroup actions over a set, semi-direct products, inverse semigroups and their extension to semigroups with local units (called resettable semigroup in the present paper).

FARM'17, September 9, 2017, Oxford, UK

ACM ISBN 978-1-4503-5180-5/17/09…$15.00
https://doi.org/10.1145/3122938.3122943

Though well-known in semigroup theory, these concepts are yet quite unknown in programming language theory[1] despite the fact they perfectly fit, as shown below, our needs.

Structure of the paper. We start our presentation by examining the semantics of Logo's turtle programs *in which commands for movement and drawing produce line graphics either on screen or with a small robot called a turtle. The language [..] enable [..] "body-syntonic reasoning", where students can understand, predict and reason about the turtle's motion by imagining what they would do if they were the turtle.*[2].

After reviewing a basic (equivalent) syntax for turtle programs, we show that their semantics can be described by means of a monoid. This semantic monoid is shown to be an inverse monoid [15] with elements reflecting both the moves and the drawings performed by the turtle.

The algebra theoretic concepts that support our semantic models are detailed throughout. Then we show, in the remainder of the text, how these concepts are applicable for lifting Logo's turtle program to time and additional space dimensions, i.e. for animation and 3D drawings. Little incursions into monads also provide a rather comfortable syntax with **do** notation and timely rendering of media programs within the IO monad.

Notations. Most examples, formulas and programs are given using Haskell syntax. In particular, as in most functional programming language, we write $f\ x$ for the application of the function f to the argument x. Also, our paper makes intensive use of the notion of semigroups: sets equipped with a associative binary operator. The semigroup binary operator is denoted by $\diamond$. We also use the notion of monoids: semigroups with a neutral element, denoted by *mempty*, with the binary operator possibly denoted by *mappend* when used as a function, and the concatenation of monoid elements in a list denoted by *mconcat*.

2 The Turtle Example

Our approach can be illustrated in a fairly simple way by analyzing the semantics of basic logo programs that allow for defining pictures by commanding turtle movements. These programs are, from now on, called turtle programs.

It is an easy observation that turtle programs act both on the space of possible pen positions and the figures that are drawn. Together, positions and figures form the states that are eventually transformed by programs.

Actions on positions are called moves, actions on figures are called drawings. Together they form the turtle program semantic model we aim at defining.

It occurs that this simple drawing language offers a deep insight into the mathematically well structured semantics we propose for temporal media programming language.

2.1 Program Syntax

Turtle programs are defined by means of series of basic actions. Up to a minor technical change from the original definition[3], basic actions are defined by

data *BasicAction d* = *TogglePen* | *Turn d* | *Walk d*

which respectively allows for flipping up or down the pen's head (*TogglePen*), pivoting by the specified angles (*Turn*) or moving forward by the specified distance (*Walk*). Turtle programs are then defined as lists of basic actions.

type *Program d* = [*BasicAction d*]

Example. With angles defined in degree and provided we start with the pen down, the series of actions *Walk* 1, *Turn* 90, *Walk* 1, *Turn* 90, *Walk* 1, *Turn* 90, *Walk* 1, *Turn* 90, draws a triangle with unitary length borders as depicted in Figure 1 where a final *TogglePen* action has also been performed. In this figure, each move has been grouped with its following turn.

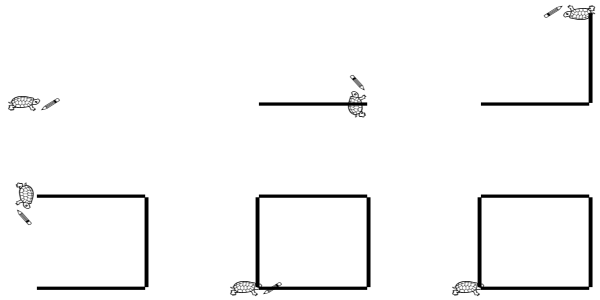

Figure 1. The turtle square.

Remark. For the sake of simplicity, we use the same type d as both the type distance in a walk or the type angle in a turn. Later in the text, the same type d is even used as the type of time duration. Of course, distinguishing all these types would certainly lead to a more constrained hence a safer type system.

2.2 Moves, Drawings and Semantics Types

Defined by lists of basic actions, turtle's programs form a monoid where sequential composition of program is modeled by list concatenation. Aiming at defining a compositional semantic model for these programs, we thus look for a semantic monoid.

The turtle performs, both at the same time, moves when changing positions, and drawings when creating new figures. They both constitute the elements of our semantics.

Formally, a turtle position is defined as a triple composed of a boolean (true when the pen is in drawing position, false otherwise), a 2D vector (the pen coordinates in the plane), and an angle (the turtle orientation). Moves are defined as position transformations.

type *Position d* = (*Bool*, V_2 *d*, *d*)
type *Move d* = *Position d* → *Position d*

[1] with noticeable exceptions as detailed at the end of the paper
[2] From Logo article on Wikipedia

[3] We replace both PenDown and PenUp (irreversible) commands of Logo by a single (reversible) TogglePen command. This is just a technical point since PenUp and PenDown command can also be turned into reversible commands provided their history is recorded in Turtle states.

A figure is defined by a set of drawn segments, that is, a set of pairs of points. Since the turtle never reads a picture, but, instead, produces a picture from a given position, drawings also take positions as input.

```
type Figure d = Set (V2 d, V2 d)
type Drawing d = Position d → Figure d
```

Program states are defined as pairs of positions and figures, and, program semantics as pairs of moves and drawings.

```
type State d = (Position d, Figure d)
type Semantics d = (Move d, Drawing d)
```

Clearly, the type *Semantics d* that is given above is not a functional type as one could expect. However, turning a program semantics into a program that draw figures, or into a function over states that compute a new state can be done as follows.

```
runS :: Semantics d → Position p → Figure d
runS (_, d) p = d p
applyS :: Semantics d → State d → State d
applyS (m, d) (p, f) = (m p, f ◇ d p)
```

with figures forming an idempotent and commutative monoid with union as product and empty set as neutral element.

Producing a figure from a given position, as done by function *runS*, just amounts to applying the drawing function specified by the second component of the semantic model.

Applying a semantics to a program state, as done by function *applyS*, is a little more tricky. The output position is computed by applying the move part of the semantics m to the input position p. The output figure is computed by merging the input figures f with the new figure computed by applying the drawing part of the semantics d to the input position p.

Remark. We aim at making the control structures, the functional dependencies and the algebraic properties of the underlying semantics explicit: a much less obvious task. The generalization of these constructions, presented below, may look uselessly (or even pedantically) theoretical. However, it has a merit: as we shall see, it generalizes to a fairly wide range of temporal media programs.

2.3 Moves, Drawings and Semantic Monoids

As already mentioned, we are looking for a semantic monoid. It occurs that the types above lead to the definition of three monoids instead of one. The third one, the program semantic monoid, is obtained by combining uniformly the first two, the moves and the drawings monoids.

First, we observe that both moves and drawings form monoids[4].

```
instance Monoid (Move d) where
  mempty = id
  mappend = flip (∘)
instance Monoid (Drawing d) where
  mempty = const mempty
  mappend d1 d2 = λp → d1 p ◇ d2 p
```

With the help of the above monoids, program semantics, that is, pairs of moves and drawings, should be sequentially combined as follows:

```
combine :: Semantics d → Semantics d → Semantics d
combine (m1, d1) (m2, d2) = (m1 ◇ m2, d1 ◇ (d2 ∘ m1))
```

On the move side, the sequential application of two turtle programs combines the moves defined by these programs. On the drawing side, the above expression says that, from a given input position p, a first figure $d_1\ p$ is drawn by the first program, moving the turtle to a position $m_1\ p$ from which a second figure $d_2\ (m_1\ p)$ is drawn. This is, indeed, the expected sequential composition of semantics we seek for.

The function *combine* turns the type *Semantics d* into a monoid by

```
instance Monoid (Semantics d) where
  mempty = (id, const mempty)
  mappend = combine
```

Then, one can check that, for every turtle semantics s_1 and s_2, we have

$$applyS\ (s_1 \diamond s_2) == (applyS\ s_2) \circ (applyS\ s_1)$$

This follows from the similarities one can observe between our definition of the *applyS* and the *combine* functions.

2.4 Basic Moves and Drawings Implementation

So far, we have defined types and combinators for composing semantics of turtle programs. These types are based on the decomposition of our semantic values into moves and drawings. It thus remains to explicit the moves and drawings associated to basic actions in order to recover a complete turtle semantic model.
Basic moves semantics is defined as follows.

```
moveA :: BasicAction d → Move d
moveA TogglePen (b, v, a) = (¬ b, v, a)
moveA (Turn d) (b, v, a) = (b, v, a + d)
moveA (Walk d) (b, v, a) = (b, v + w, a)
  where w = (V2 (d ∗ cos ra) (d ∗ sin ra))
        ra = 2 ∗ pi ∗ a/360
```

The move semantics associated to every program, that is, list of basic actions, is then defined by:

```
move :: Program d → Move d
move prog = mconcat (map moveA prog)
```

As expected, the semantics of a sequential composition of programs is mapped to the (flipped) composition of their semantics. Basic drawing semantics is defined similarly.

```
drawA :: BasicAction d → Drawing d
drawA (Walk d) (False, _, _)    = mempty
```

[4] Many instances defined in this pages clash with Haskell Prelude instances or even one on another. In our real implementation **newtype** embeddings are used. However, for the sake of clarity, they have been omitted throughout, as well as have been omitted most type constraints.

```
drawA (Walk d) p@(True, v, _) = insert (v, nv) mempty
  where (_, nv, _) = moveA (Walk d) p
drawA _ _ = mempty
```

The drawing semantics of program is defined as the second projection of the complete semantics of programs.

```
draw :: Program d → Drawing d
draw prog = snd (sem prog)

semA :: BasicAction d → Semantics d
semA prog = (moveA prog, drawA prog)

sem :: Program d → Semantics d
sem prog = mconcat (map semA prog)
```

Observe that drawing semantics is slightly more complex to define than move semantics because it depends on positions which may be altered by moves. Nevertheless, this corresponds to a fairly well known construction in algebra called semi-direct product that is detailed below.

This semantics is compositional in the sense that, for all program *p1* and *p2*, we have

$$\begin{aligned} sem\ mempty &== mempty \\ sem\ (p1 \diamond p2) &== sem\ p1 \diamond sem\ p2 \end{aligned}$$

Indeed, the function *sem* is the unique monoid morphism from the (free) monoid of turtle programs into turtle semantics that is induced by (*moveA*, *drawA*) on elementary actions.

2.5 Remarks on Basic Turtle Moves

At first sight, one may think that a turtle move essentially traverses the 2D space defined by the (x,y)-coordinates of every position. However, the capacity to turn says that the turtle can also act on the space itself: namely its underlying 2D basis.

Indeed, by combining basic turtle walks and turns, one can define any isometric and orientation preserving transformation of the underlying 2D space, that is, the transformations generated by arbitrary translations and rotations.

Extended with additional basic moves, turtle semantics captures arbitrary isometries (adding reflexion across, say, the x-axis), arbitrary affine bijections (adding reflexion and non zero contraction/walking speed along the x-axis) or even arbitrary affine transform (also adding projection of the entire 2D space into a line by means of zero scaling along the x-axis).

This observation plays a key role when extending turtle moves to additional space or time dimensions.

3 Monoid Semantics Revisited

We have seen above that monoids (or even just semigroups) play a central role in the definition of the turtle semantics. Moreover, most semantics definitions are generic. Functions *move*, *draw* and *sem* are uniformly defined in terms of basic actions (*BasicAction d*) and their move and drawing semantics (functions *moveA* and *drawA*).

This suggests that the turtle semantics relies on some general structures that are worth being made explicit. In this section, we study more in depth the generic constructions that have been (yet implicitly) applied and the rather rich mathematical properties these constructions satisfy.

3.1 Semi-direct Product

We first observe that the turtle semantic monoid is a fairly standard construction known in semigroup theory as the semi-direct product of two monoids.

This notion is defined via what algebraists call an action of a monoid over a set.

```
class (Monoid m) ⇒ MonoidAction m a where
  act :: m → a → a
```

with the requirement that properties should hold for all a, m_1 and m_2 of adequate types.

$$act\ mempty\ a == a \quad (1)$$

$$act\ m_1\ (act\ m_2\ a) == act\ (m_1 \diamond m_2)\ a \quad (2)$$

The associated notion of semi-direct product of monoids is defined by:

```
instance (MonoidAction m a, Monoid a) ⇒ Monoid (m, a)
  where
    mempty = (mempty, mempty)
    mappend (m1, a1) (m2, a2) =
      (m1 ◇ m2, a1 ◇ act m1 a2)
```

Back to the turtle semantics, we simply put:

```
instance MonoidAction (Move d) (Drawing d) where
  act m d = d ∘ m
```

The generic monoid instance given above applies to *Semantics d*, turning it into a monoid and we have *combine* == (◇).

3.2 Inverse Monoid

So far, we haven't used the fact that (turtle) moves are reversible. It turns out that those properties open the way to a transformation of turtle program, called *inverseP* that have, up to semantic equivalence, fairly interesting properties.

```
inverseP :: Program d → Program d
inverseP p = map (invA) (reverse p)
  where invA TogglePen = TogglePen
        invA (Turn d) = Turn (−d)
        invA (Walk d) = Walk (−d)
```

Then, one can check that, for every program p, programs p and $p \diamond inverseP\ p \diamond p$ have the same semantics, as well as programs *inverseP p* and $inverseP\ p \diamond p \diamond inverseP\ p$. In semigroup theory, *inverseP p* is called the semigroup inverse of p. This leads us to the definition of the class of inverse semigroups.

```
class Semigroup m ⇒ InverseSemigroup m where
  inverse :: m → m
  norder :: (Eq m) ⇒ m → m → Bool
  norder m n = m == (m ◇ inverse m ◇ n)
```

with the requirement that the following properties should hold for arbitrary elements a and b of the semigroup m.

$$a \diamond inverse\ a \diamond a \quad == \quad a \tag{3}$$

$$inverse\ a \diamond a \diamond inverse\ a \quad == \quad inverse\ a \tag{4}$$

which states, following inverse semigroup theory [15], that $inverse\ a$ is the semigroup inverse of a, and

$$\begin{aligned} & a \diamond inverse\ a \\ & \quad \diamond b \diamond inverse\ b \quad == \quad b \diamond inverse\ b \\ & \qquad\qquad\qquad\qquad \diamond a \diamond inverse\ a \end{aligned} \tag{5}$$

which states that elements of the form $a \diamond inverse\ a$ commute.

Then, inverse semigroup theory ensures that idempotents[5], necessarily of the form $a \diamond inverse\ a$, commute and that inverse are uniquely defined, i.e. $inverse\ a$ is the unique element b such that both properties $a \diamond b \diamond a == a$ and $b \diamond a \diamond b == b$ hold.

As another derivative of inverse semigroup theory, the relation *norder* is a partial order relation known as the *natural order* [17]. The word *natural* refers in semigroup theory to the fact it is uniformly defined over all inverse semigroups.

Then we can show that turtle programs form, up to semantic equivalence, an inverse monoid:

```
instance InverseSemigroup (Program d) where
  inverse = inverseP
```

with properties (3)–(5) valid up-to semantic equivalence. From the semantics side, given

$$(m, d) = sem\ p \quad \text{and} \quad (m', d') = sem\ (inverse\ p)$$

for some turtle program p, one can check that we have

$$m' \circ m == id == m \circ m' \quad \text{and} \quad d' == d \circ m'$$

This confirms that, as already mentioned, the moves induced by Turtle programs are in fact bijections, or, stated differently, the moves induced by turtle programs form a group.

Then, it is a known fact in semigroup theory (see [15], Section 7.1) that a sufficient condition for the semi-direct product (m, a) of a monoid m by a monoid a to produce an inverse semigroup is that m is in fact a group and a is a lattice with product as meet. This is the case for (the subtype of bijective) moves and drawings induced by turtle programs. This fact concludes the proof that the program monoid is, up to semantic equivalence, an inverse monoid.

3.3 Resettable Monoid

The inverse semigroup detailed above also induces an additional program transformation, called the *reset* operation, that is also of interest. It is definable even in the case of non reversible moves, and, as we shall see, it behaves like a fork operator hence allowing concurrent evaluation.

More precisely, we define the function *resetP* over turtle programs by

```
resetP :: Program d → Program d
resetP p = p ◇ inverse p
```

[5]The elements b such that $b \diamond b == b$

When executing the program $p \diamond inverse\ p$, the turtle performs the actions specified by p, producing some figure, and then it executes the inverse of these actions in the reverse order, reproducing the same exact figure while moving back to the starting position. In other words, the program *resetP p* describes a double drawing of the figure defined by p; a forward drawing with p and a backward drawing with $inverse\ p$.

At the semantic level, such a redundancy can be avoided as follows. Given the semantics

$$(m, d) = sem\ p$$

of the program p, that says that the turtle should move following m while drawing following d, one can check that we have

$$(mempty, d) == sem\ (resetP\ p)$$

In other words, the program *resetP p* can also be read as: from any given initial position, send a copy of the current turtle to perform the expected drawing and disappear upon termination while the initial turtle stays in that position waiting for further instructions.

This says at least two things. First, the reset function acts over a program p as a sort of fork operation that allows parallel evaluation. Second, implementing the reset directly at the semantic level is more efficient and can be defined even in the case of non reversible moves.

Generalizing these observations leads us to the definition of resettable semigroups.

```
class (Semigroup m) ⇒ ResettableSemigroup m where
  reset :: m → m
```

with the requirement that the properties should hold for arbitrary elements a and b of the semigroup m.

$$reset\ a \diamond a \quad == \quad a \tag{6}$$

$$reset\ a \diamond reset\ a \quad == \quad reset\ a \tag{7}$$

$$reset\ a \diamond reset\ b \quad == \quad reset\ b \diamond reset\ a \tag{8}$$

$$reset\ b \diamond a == a \quad \Rightarrow \quad reset\ b \diamond reset\ a == reset\ a \tag{9}$$

In semigroup theory, resettable semigroups are known as *left semi-adequate semigroups* (see [5] and the related works mentioned at the end). Property (6) says that $reset\ a$ is a left local unit of a. Properties (7) and (8) say that these left local units are idempotent and commute henceforth form a semi-lattice with product as meet. Property (9) says that $reset\ a$ is the least left local unit of a in this semi-lattice. From these properties, one can also prove that the property $reset \circ reset == reset$ is satisfied. In other words, the *reset* function is a projection.

Indeed, let a be some monoid element, let $e = reset\ a$ and let $e' = reset\ e$. It remains to prove that $e == e'$. By property (6) we have $e' \diamond e == e$. This implies that $e \leqslant e'$ in the semi-lattice of local units. But since e is idempotent, we have $e \diamond e = e$ hence, by property (9) $e' \leqslant e$. Together, this implies that $e == e'$.

Of course, every monoid can be turned into an instance of *ResettableSemigroup* by taking $reset = const\ mempty$. However, this instance is of very little interest. Incidentally, we do not require m to be a monoid as shown by the premises $Semigroup\ m$ of the class definition. In the absence of a neutral element, existence of a *reset* is by no mean a trivial property.

Inverse semigroups are particular cases of resettable semigroups thanks to the following (non trivial) instance:

```
instance InverseSemigroup m ⇒ ResettableSemigroup m
  where
    reset m = m ◇ inverse m
```

and, as already observed above, over semi-direct product, one may prefer the following and more efficient instance.

```
instance (SemigroupAction m a, Semigroup a) ⇒
  ResettableSemigroup (m, a)
    where
      reset (_, a) = (mempty, a)
```

Applied to turtle semantics, this can be made explicit by:

```
instance ResettableSemigroup (Semantics d) where
  reset (_, d) = (mempty, d)
```

Of course, there is no syntactic way to reset efficiently a turtle program. But, as already noticed, thanks to our uniform semantics, this is really just a matter of syntax. We can add a basic action *Reset* (*Program d*) that takes an entire program as argument and whose semantics is defined by *moveA* (*Reset* _) = *mempty* and *drawA* (*Reset p*) = *draw p*.

Also, we do not use here the fact that the set of moves induced by turtle programs are reversible. This means that we may even add irreversible basic move commands such as *WalkToPosition v*, a command that is irreversible, without losing the possibility to perform a reset.

4 Induced Monad Semantics

So far, our turtle programming language is essentially based on lists over some parametrized alphabets (basic actions) with mathematically rich but not (yet) clearly usable properties. For instance, conditionals, loops and variable bindings are missing for drawing complex figures. Also, effectively drawing figures necessitates IO actions. These leads us to embed the turtle semantics into monads.

4.1 Drawing Monad Semantics

Embedded in Haskell, our proposal inherits all the programming framework proposed by Haskell. However, combining Haskell functions and turtle programs may not be that convenient. The monoid product syntax (◇) does not sound much like programming at all.

Also, the above discussion about the reset function and possible extension to non reversible moves suggests that our programs should be defined directly at the turtle semantics level as pairs of moves and drawings with potential unnecessarily heavy turtle state transfer.

The monad approach (and the do notation) offers a clean and efficient approach for this purpose. It occurs that Haskell GHC Prelude even offers an easy way to embed a monoid into a monad.

The following instance, recalled below for the sake of reasonable self completeness, can relevantly be used:

```
instance Functor ((,) a) where
  fmap f (x, y) = (x, f y)
instance Monoid a ⇒ Applicative ((,) a) where
  pure x = (mempty, x)
  (u, f) <*> (v, x) = (u `mappend` v, f x)
instance Monoid a ⇒ Monad ((,) a) where
  (u, a) >>= k = case k a of (v, b) → (u `mappend` v, b)
```

with $\gg=$ the bind operator of monads. Our turtle semantics is then embedded into a monad by defining:

```
type MSemantics d a = (Semantics d, a)
```

This allows for (re)defining basic moves as monadic actions:

```
nop = load mempty
toggle = load [TogglePen]
walk d = load [Walk d]
turn d = load [Turn d]
```

with the loading and running function defined by:

```
load :: Program d → MSemantics d ()
load p = (sem p, ())
runM :: MSemantics d a → Figure a
runM ((_, d), _) = d (True, 0, 0)
```

Then, turtle programs inherit from monad do notation.

Example. The following program draws a triangle.

```
prog = do { walk 1; turn 120; walk 1; turn 120; walk 1 }
triangle = runM prog
```

4.2 Drawing IO-monad Semantics

Viewing figures on the screen necessitates an additional function *render* :: *Figure d* → *IO* () which code, depending on the chosen graphical library, is not detailed here.

It follows that turtle semantics can instead be seen as an IO action that takes a position as input and actually produces a drawing while also sending back the new current position. The type of moves over a space type *s* can thus be described by:

```
type IOSemantics s a = s → IO (s, a)
```

with the *Monad* instance

```
instance Monad (IOSemantics s) where
  return a s = return (s, a)
  (>>=) m f s = do
    (ns, a) ← m s
    (f a) ns
```

Then, thanks to the *render* function, one can lift turtle monoid semantics into IO semantics by:

```
liftS :: Semantics d → IOSemantics (Position d) ()
liftS (m, d) p = do
  render (d p)
  return (m p, ())
```

or turtle monad semantics into IO semantics by:

```
liftM :: MSemantics d a → IOSemantics (Position d) a
liftM ((m, d), a) p = do
  render (d p)
  return (m p, a)
```

Various equalities could be written down in order to validate all these definitions. For instance, one can check that we have

$$liftM\ (load\ p) == liftS\ (sem\ p)$$

for all turtle program p. The important point is that the monoid, the monad or the IO monad approaches essentially define equivalent semantics.

Remark. Command by command rendering may be useful in programming context such as live coding. However, for proper 2D or even 3D rendering of complex pictures, it may be worth accumulating in the underlying state all drawing commands so that, ultimately, after evaluating an entire program, features that are context dependent, such as face covering in 3D drawings, can be computed and rendered correctly.

5 Time Extension

So far, our turtle is capable of performing figures in a two dimensional space, henceforth performing 2D drawings. We aim now at extending our turtle with time in order to define 2D animation. All the algebraic tools developed so far can be used for this purpose.

5.1 Animation

At the semantics level, extending our turtle to the time dimension is rather easy. We just define the type:

```
type Animation d = d → Semantics d
```

The meaning of an element $a :: Animation\ d$ is the following: the move performed and the image drawn at some time stamp t is given defined by $a\ t$. In other words, we can define the function

```
runA :: Animation d → d → Position d → Figure d
runA a t = runS (a t) p
```

that simply draws the figure specified by the animation at every instant.

Since *Semantics d* is known to be a monoid, animations can also be turned into a monoid by letting

```
instance Monoid (Animation d) where
  mempty = const mempty
  mappend a1 a2 = λt → (a1 t) ◇ (a2 t)
```

Just given as a reminder as this is a particular instance of the generic monoid instance of the monoids of functions into a monoid.

Example. Every figure f can simply be embedded into an animation by defining $liftA\ f = const\ (id, const\ f)$, that is, the turtle programming displaying the figure f at every instant and from any position. Then we have

$$liftA\ (f_1 \diamond f_2) == liftA\ f_1 \diamond liftA\ f_2$$

Observe that, with arbitrary animations and turtle moves, the monoid composition of two animation defined above is non trivial. It is not a mere superposition of two animations.

Remark. The figure produced by $(a_1 \diamond a_2)\ t$ at every instant t, is not in general the superposition of the two figures produced by $a_1\ t$ and $a_2\ t$. The space position from which the drawing induced by $a_2\ t$ is performed is indeed defined by the position where the turtle stops when drawing $a_1\ t$.

It turns out that composing two animations from the same position necessitates a reset that can be defined by

```
instance ResettableSemigroup (Animation d) where
  reset a = λt → reset (a t)
```

that inherits from the *ResetableSemigroup* instance of the type *Semantics d*. The timed superposition of two animations a_1 and a_2 at the same instant and position can then be defined by $reset\ a_1 \diamond a_2$.

5.2 Temporal Moves

In terms of semantics, the animation type defined above already allows for defining any 2D animation. However, in terms of temporal programming, one may be in need of uniformly defined functions that allows to start, stop or run an animation from, to or between two specified timestamps. One may also wish to modify the speed of a given animation. Even more importantly, one may feel the need to combine animations over time.

This observation leads us to the definition of the temporal moves type and its associated monoid instance:

```
type TMoves d = d → Maybe d
instance Monoid (TMoves d) where
  mempty = Just
  mappend f g = λt → case (f t) of
    Just t1 → g t1
    Nothing → Nothing
```

that generalizes the flipped function composition.

Remark. The usage of the type *Maybe d* as codomain may look a bit awkward at first sight. However, this turns out to be the most elegant way to "block" passing time while preserving the property that an instant has zero duration. Indeed, when blocking time, no instant value will be given.

Examples. First examples of timed moves are delay functions of the form $\lambda t \to Just\ (t + d)$ for some constant value d. In other words, this function allows for modeling the fact that a turtle move may last d unit of time.

The (monoid) composition of delays $\lambda t \to Just\ (t + d_1)$ and $\lambda t \to Just\ (t + d_2)$ equals delay $\lambda t \to Just\ (t + d_1 + d_2)$. As expected, this reflects the fact that the two delay values have been summed.

Many other examples of timed moves can be defined such as, for instance, the time stretch $\lambda t \to Just\ (d * t)$ that amounts to compressing (when $d < 1$) or stretching (when $d > 1$) the time dimension by some (generally strictly positive) constant.

However, in the absence of "visible" effect of these timed moves over animations makes these examples a bit abstract. The real power

of timed moves appears when associated with animations as shown right below.

5.3 Temporal Semantics Extension

The temporal extension of the turtle semantics type is obtained by pairing timed moves and animations.

```
type TExtension d = (TMoves d, Animation d)
```

The monoid of timed moves acts over animations by

```
instance MonoidAction (TMoves d) (Animation d) where
  act d a = λt → case (d t) of
    Just t1 → a t1
    Nothing → mempty
```

where, in the case time is stopped by a temporal move, the animation no longer produces any figure but the empty one.

Then, following the definition of semi-direct product construction, we have:

```
instance Monoid (TExtension d) where
  mempty = (mempty, mempty)
  mappend (d1, a1) (d2, a2) = (d1 ◇ d2, a1 ◇ act d1 a2)
```

all these instances given just as a reminder. Reset is extended to timed semantics (and the underlying animations) by the following instance:

```
instance ResetableSemigroup (TExtension d)
  where reset (_, a) = (Just, reset a)
```

which amounts to superpose two (embedded) animations at the same time and position. One can check that we have

$$reset\ (d_1, a_1) \diamond (d_2, a_2) == (d_2, reset\ a_1 \diamond a_2)$$

for all time moves d_1 and d_2 and all animations a_1 and a_2.

Running a timed semantics from a given instant t and a given space position p can be defined by the function

```
runTS :: TExtension d → d → Position d → Figure d
runTS (_, a) t p = runA a t p
```

which explicits the fact that the temporal move d in the timed semantics (d, a) only affects the animations that will be combined to the right of the animation (d, a). This generalizes to the time dimension the fact that spatial move performed by a given 2D turtle program p affects the drawings performed by programs combined to the right of p as illustrated by the following examples.

Every animation a can simply be embedded into a timed semantics by defining

```
liftTS :: Animation d → TExtension d
liftTS a = (Just, a)
```

Then we have

$$liftTS\ (a_1 \diamond a_2) == liftTS\ a_1 \diamond liftTS\ a_2$$

for all animation a_1 and a_2.

Examples. As a first application example, delaying an animation a by d units of time can be defined by $delay\ d \diamond liftTS\ a$ with $delay\ d = (\lambda t \rightarrow Just\ (d + t), mempty)$.

As another application example, stretching an animation by a non-zero factor f can be defined by $stretch\ f \diamond liftTS\ a$ with $stretch\ f = (\lambda t \rightarrow Just\ (f * t), mempty)$.

Gathering these timed semantics elements that only act on the time dimension eventually leads us to the definition of the following class type.

5.4 Timed Monoid Class Type

The class type *TimedMonoid* is defined by:

```
class (Monoid m) ⇒ TimedMonoid m d | m → d where
  liftTM :: (d → Just d) → m
  delay :: d → m
  delay d = liftTM (Just ∘ (d+))
  stretch :: d → m
  stretch d = liftTM (Just ∘ (d*))
  start :: m → m
  stop :: m → m
  play :: d → d → m → m
  play t1 t2 m = start (delay (t1 − t2)
   ◇ stop (delay (t2 − t1) ◇ m))
```

In the class type defined above, the method *liftTM* method shall lift any time transformation into the timed monoid m in such a way that the derived methods *delay* and *stretch* behave as already illustrated in the examples above.

Methods *start* and *stop* shall start or stop a timed process at a given (fixed) time origin, say, the zero instant. The derived method *play* then plays a timed element from the instant t_1 to the instant t_2.

Timed moves form an instance of timed monoid:

```
instance TimedMonoid (TMove d) d where
  liftTM = id
  start d t = if (t < 0) then Nothing else d t
  stop d t = if (t > 0) then Nothing else d t
```

The timed monoid instance of timed semantic extension follows from the above as it uniformly derives from the timed monoid instance of timed moves[6].

```
instance TimedMonoid (TExtension d) d where
  liftTM d = (d, mempty)
  start (d, a) = (d, act (start mempty) a)
  stop (d, a) = (d, act (stop mempty) a)
```

Example. Running the timed semantics *play* 0 *d* (*liftTS f*) will display the figure f from the instant 0 to the instant d, inclusive, performing no drawings when outside this interval.

[6] The *ScopedTypeVariables* option, necessary to remove any ambiguity on the type of *mempty* has been omitted in this instance.

Remark. It can be shown that the monoid of timed semantics generated by delays, non-zero stretches, starts and stops, and arbitrary lifts of animation is not an inverse monoid.

It could be turned into one by propagating time cuts induced by start and stop. However, such a propagation of time cuts does not seem to be desirable. On the contrary, the product of timed semantics computes instead some kind timed envelope of the combined animation much like in the way spatial envelopes are computed in Diagrams [19].

Example. There is the notion if partial identities over d that can be defined as function $p :: d \to \mathit{Just}\ d$ such that if $p\ t = \mathit{Just}\ t'$ for some $t :: d$ then $t = t'$, i.e. seen as a partial function, a partial identity is essentially defined by its domain.

Then, as an example of implicit computation of temporal envelopes, one can check that for any partial identities d_1, d_2, for all animation a_1, a_2, we have $(d_1, a_1) \diamond (d_2, a_2) == (d, a_1 \diamond a_2)$ where d is the partial identity (uniquely) defined by the union of the domain of d_1 and d_2.

6 Space Extension

So far, our turtle is spatio-temporal. It performs timed 2D figures made of curves. Now we aim at extending its space to perform 3D figures made of surfaces. Again, most of the algebraic tools developed above are reusable for this purpose.

6.1 Extruding 3D Turtle

There are many possibilities for defining a 3D drawing language. Following the idea that drawings should be achieved by means of combining turtle pen position and moves one possibility is to define 3D surface by mean of extrusion: a well known 3D technique that aims to create 3D surfaces by moving in the space some varying 2D curve.

More precisely, in the 2D case, we observe that figures are drawn by "extruding" turtle positions that defines points (when pen is down) into segments. Extending turtle positions to 3D space with basic 2D curves as possible turtle pen shapes eventually leads us to the desired programming language.

As preliminary code, we define 3D affine transform by

```
type Affine3D d = (M33 d, V3 d)
instance Monoid (Affine3D d) where
  mempty = (V3 (V3 1 0 0) (V3 0 1 0) (V3 0 0 1), 0)
  mappend (m1, v1) (m2, v2) = (m1 !*! m2, v1 + m1 !* v2)
```

Then, limiting our example to circle shapes with a given radius, turtle pen shape, basic 3D actions and 3D turtle programs are defined by

```
data Penshape d = Empty | Circle d
data BasicAction3D d =
  Pen (Penshape d) |
  Transform (Affine3D d)
type Program3D d = [BasicAction3D d]
```

3D turtle moves are defined by arbitrary affine transformation of the underlying 3D space (command *Tranform*). Extrusion is performed between two successive non empty *Pen* command, separated by one or more *Transform* command that are implicitly combined. This allows for defining complex movements by composition of simpler ones, realizing the extrusion only when the resulting complex movement is defined.

Example. As program examples, we define

```
circle :: d → Program3D d
circle d = [Pen (Circle d)]
shift :: (d, d, d) → Program3D d
shift (x1, x2, x3) = [Transform (0, V3 x1 x2 x3)]
cylinder :: d → Program3D d
cylinder r l = circle 0 ◇ circle r ◇
  shift (0, 0, l) ◇ circle r ◇ circle 0
```

Then, as it shall become clear with the extrusion semantics detailed below, the expression *cylinder r l* indeed specifies a cylinder with radius r and length l along the z-axis.

6.2 3D Position and Scene

Three dimensional positions and figures, called *Scene*, are defined by

```
type Position3D d = (Penshape d, Affine3D d, Affine3D d)
initialPosition3D = (Empty, mempty, mempty)
type Scene d = Set (V3 d, V3 d, V3 d)
```

where a position describes the last pen shape, its associated affine transform and the current affine transform, and a scene describes a set of triangles. Again, scenes inherit from the monoid structure of sets with union as product and empty set as neutral element.

6.3 3D Moves, Drawings and Semantics Types

Three dimensional moves, drawings are defined much like in the 2D case:

```
type Move3D d = Position3D d → Position3D d
instance Monoid (Move3D d) where
  mempty = id
  mappend = flip (∘)
type Drawing3D d = Position3D d → Scene d
instance Monoid (Drawing3D d) where
  mempty = const mempty
  mappend d1 d2 = λp → (d1 p ◇ d2 p)
```

with a similar semigroup action and derived semi-direct product for semantics, defined by:

```
instance SemigroupAction (Move3D d) (Drawing3D d)
  where
    act m d = d ∘ m
type Semantics3D d = (Move3D d, Drawing3D d)
instance Monoid (Semantics3D d) where
  mempty = (mempty, mempty)
  mappend (m1, d1) (m2, d2) = (m1 ◇ m2, d1 ◇ act m1 d2)
```

6.4 Running 3D Turtle

As for running is concerned, we define the move and drawing semantics for 3D turtle basic actions and programs by:

```
move3DA :: BasicAction3D d → Move3D d
move3DA (Pen ps) =
  λ(_, _, aff) → (ps, aff, aff)
move3DA (Transform aff) =
  λ(ps, p, aff0) → (ps, p, aff0 ◇ aff)
move3D :: Program3D d → Move3D d
move3D p = mconcat (map move3DA p)
draw3DA :: BasicAction3D d → Drawing3D d
draw3DA (Pen Empty) = (const mempty)
draw3DA (Pen (Circle d)) = (extrude d)
draw3DA _ = mempty
draw3D :: Program3D d → Drawing3D d
draw3D p = let (_, d) = sem3D p in d
sem3D :: Program3D d → Semantics3D d
sem3D p = mconcat
  (map (λa → (move3DA a, draw3DA a)) p)
```

with the *extrude* function defined by

```
extrude :: d → Position3D d → Scene d
extrude _ (Empty, _, _)
  = mempty
```

in the case the last turtle pen position is empty,

```
extrude 0 (Circle 0, _, _)
  = mempty
```

in the case the last and current turtle pen position have zero radius hence both define a point[7]

```
extrude d (Circle 0, (_, v), aff)
  = span3D v (d, aff)
```

in the case the previous turtle pen shape has zero radius hence define a single point,

```
extrude 0 (Circle d, aff, (_, v))
  = cospan3D (d, aff) v
```

in the case the current turtle pen shape has zero radius hence define a single point, and

```
extrude d2 (Circle d1, aff1, aff2)
  = tube3D (d1, aff1) (d2, aff2)
```

in the case both last and current turtle pen shapes are ellipse with non zero radius.

[7] hence inducing only a segment that is omitted

Functions *span3D*, *tube3D* and *cospan3D* allow for triangulating the surface defined by extrusion from one position to another. Their semantics is informally depicted in Figure 2.

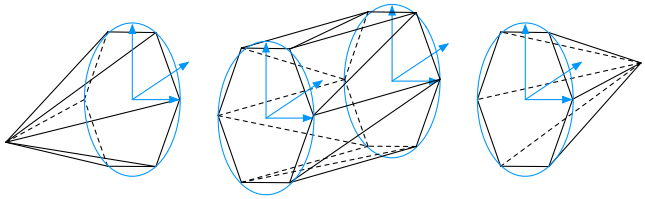

Figure 2. Triangulation examples respectively performed by functions *span3D*, *tube3D* and *cospan3D*.

6.5 The Octopus Experiment

Our current implementation, called Octopus[8], is thus based on curve-to-curve extrusion.

Technically, our proposal is a lot more evolved than what is described above. Octopus programs, implemented in Haskell and executed on the central processing unit (CPU), define 3D scene specifications that consists in trees of (2D shaped) pen positions. Extrusion, tessellation, normal computation and lighting are executed in parallel on the graphics processing unit (GPU) via a modern OpenGL graphic pipeline with vertex, geometry and fragment shaders.

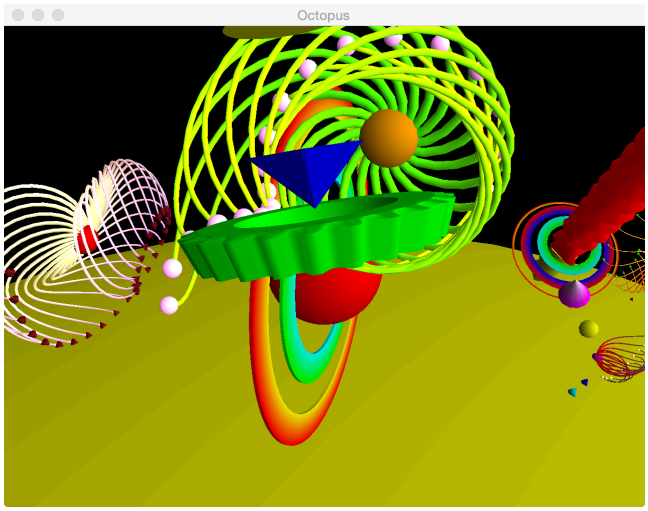

Figure 3. A scene image example.

As example, the image in Figure 3 is extracted from a 3D animated scene example, described by means of 5000 timed elements, extruded into more than 500,000 triangles per image, displayed at 50 frames per second. Its associated triangulation is given in Figure 4 and the normal colors[9] are given in Figure 5.

It must be noted that the efficiency of the rendering is also due to Haskell itself that automatically exploit the multicore structure of the CPU, easily reaching more than 180% CPU usage.

[8] see http://poset.labri.fr/octopus/

[9] The surface vertex colors are set to the vertex normal coordinates.

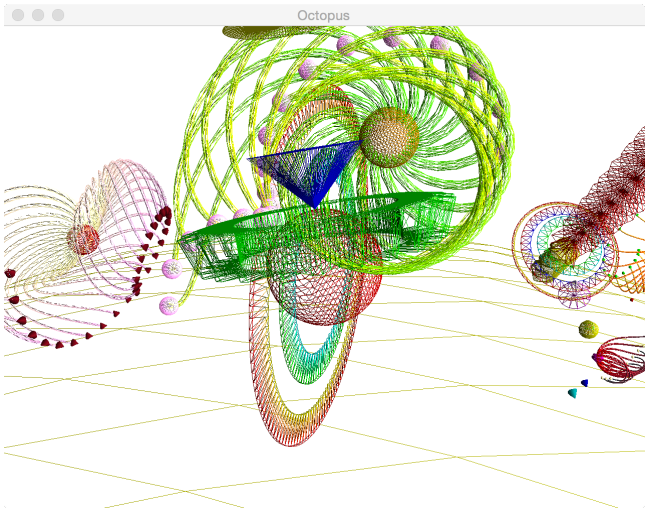

Figure 4. Associated triangulation.

Although we aim in the long term at defining a 3D animation programming language dedicated to a broad class of users, be they trained with basic geometry notions or not, this goal is yet not achieve. The Octopus language necessitates further experiments and extensions before being considered as a mature front end.

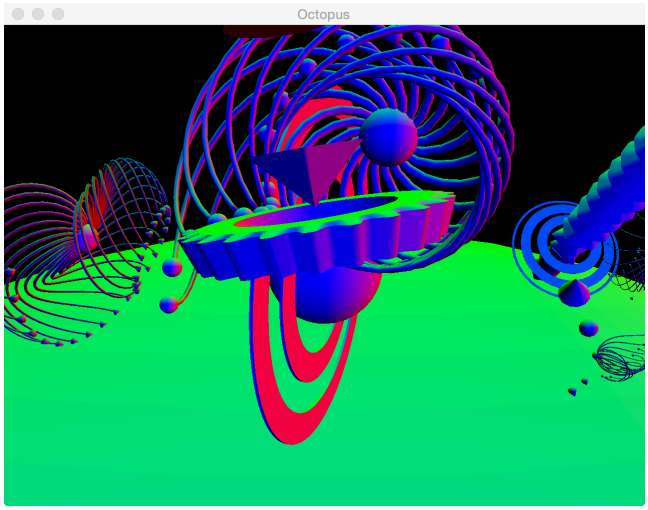

Figure 5. Associated surface normals.

A significant difference with the 2D turtle case is that, in our 3D version, a *pen* "action" creates now a drawing by extruding a surface from the pen shape and position defined by the previous *pen* "action". The moves between these two actions are thus implicitly combined to be treated as a single one.

More precisely, extrusion allows for combining several simple moves into a complex one which facilitates the task of drawing regular shapes.

For instance, a regular polygon with n vertices and (outer) radius n is defined in Logo by n repetitions of walks of $2 \times r \times sin(\pi/n)$ distance units followed by turns of $2 \times \pi/n$. Its definition necessitates some basic knowledge in trigonometry. With (2D) extrusion, it suffices instead to define $n + 1$ successive pen positions (with point shape) separated by the combined movements *Turn* 90, *Walk* r, *Turn* $(360 / n)$, *Walk* $(-r)$ and *Turn* (-90), two successive pen positions being extruded into a segment. The resulting drawing is the same without any explicit trigonometric function.

7 Related Works

Between groups and monoids, inverse semigroups have already been studied and developed quite a while in algebra[15, 16]. In computer science, they already appears in connection with Girard's Linear Logic [6] and related studies of reversible computations [1, 3]. More recently, inverse semigroup have been applied to music modeling [2, 9, 12].

Resettable monoids, first studied in [14], are known in semigroup theory as *semi-adequate* monoids [5, 14]. Although not much popular even in theoretical computer science, it is known that their free algebras capture deterministic unranked trees [13]. They have recently been studied in formal language theory by the second author for developing a new algebraic characterization of regular languages of finite trees [11]. It is conjectured that dropping the commutation hypothesis (property (8)) will eventually lead to languages of ranked trees.

To the best of our knowledge, resettable monoids have never been considered in programming language semantics even though interleaving semantics for concurrency relies for long on the notion of commuting processes : a property explicitly axiomatized in resettable monoids.

In the context of functional programming languages, drawing and animation languages have already led to many proposals. As already suggested above, the novelty of our approach does not lay our yet fairly implicit EDSL language proposal. Our front-end is still a toy language example essentially designed for studying the applicability of inverse semigroup theory to the definition of a generic and efficient back-end for media programming languages.

Nevertheless, our proposal can still be related with existing approaches among which Euterpea [8] in computer music and Diagrams [19] in computer graphics.

Indeed, both proposals provide semigroup based uniform and robust program structures that are applicable to many distinct purposes. Paul Hudak's notion of polymorphic temporal media, based on a 2D structuration of time with its (horizontal) sequential and its (vertical) parallel product is an example. The automatic computation of object envelopes of figures in Diagrams is another example. In Diagrams, semigroup actions are even made explicit (see [19] for many other examples).

Our approach clearly follows these paths already open by previous authors.

8 Conclusion

We thus propose a formal approach for defining (and implementing) a temporal media program semantics model based on semigroup theoretical concepts.

The main novelty of our approach is to make explicit the usage of semi-direct products. This allows, in particular, for combining, within semigroup structures, both (spacial or temporal) moves and (musical or graphical) renderings.

To the best of our knowledge, the notion of semi-direct product, though deriving from the notion of monoid action that is already used in Diagrams [19], have not yet been considered in temporal media programming.

Octopus, an extension of Logo's turtle for programming 3D animation is based on this approach. Efficient rendering is achieved by combining simple (but possibly large) tree shaped temporal 3D scene specifications defined in Haskell (on the CPU) with efficient rendering on a modern OpenGL graphic pipeline (on the GPU).

Though applicable to the definition of non trivial 3D animation, the resulting programming language is stable enough to be viewed as a programming language proposal. In particular, we believe that the uniform semantics modeling of both space and time dimensions that we have proposed throughout these pages is neither fully exploited not fully understood.

Our belief is that inverse (and resettable) semigroups could be much more developed towards programming semantics. For instance, the approach presented here could be generalized for combining in a fairly abstract way reversible computations, e.g. moves, with non reversible ones, e.g. drawings.

Aside the Octopus experiment, one possible development of our proposal may consist in rephrasing the FRP definitions [4] in terms of monoid constructions. Doing so, the inderlyng semigroup syntax will certainly restrict the way behaviors and events can be defined and combined, and this may eventually lead to a more efficient operational semantics. As a matter of fact, recent works on FRP (see [18]) tend to restrict FRP programs to somewhat automata (henceforth semigroup) like specifications for better efficiency in a reactive context.

Acknowledgments

Thanks to Jean-Michaël Celerier, Pierre Bénard and Mikhail Ruskin for their help with modern 3D technology.

References

[1] S. Abramsky. 2005. A structural approach to reversible computation. *Theor. Comp. Sci.* 347, 3 (2005), 441–464.
[2] S. Archipoff and D. Janin. 2016. Structured Reactive Programming with Polymorphic Temporal Tiles. In *ACM Work. on Functional Art, Music, Modeling and Design (FARM)*. ACM Press, 29–40.
[3] V. Danos and L. Regnier. 1999. Reversible, Irreversible and Optimal lambda-Machines. *Theor. Comp. Sci.* 227, 1-2 (1999), 79–97.
[4] C. Elliott and P. Hudak. 1997. Functional Reactive Animation. In *Int. Conf. Functional Programing (ICFP)*. ACM.
[5] J. Fountain, G. Gomes, and V. Gould. 1999. A Munn type representation for a class of E-semiadequate semigroups. *Journal of Algebra* 218 (1999), 693–714.
[6] P. M. Hines. 1997. *The algebra of Self-Similarity and its Applications*. Ph.D. Dissertation. University of Wales.
[7] P. Hudak. 2008. *A Sound and Complete Axiomatization of Polymorphic Temporal Media*. Technical Report RR-1259. Department of Computer Science, Yale University.
[8] P. Hudak. 2013. *The Haskell School of Music : From signals to Symphonies*. Yale University, Department of Computer Science.
[9] P. Hudak and D. Janin. 2014. Tiled Polymorphic Temporal Media. In *ACM Work. on Functional Art, Music, Modeling and Design (FARM)*. ACM Press, 49–60.
[10] P. Hudak and D. Janin. 2015. *From out-of-time design to in-time production of temporal media*. Research report. LaBRI, Université de Bordeaux.
[11] D. Janin. 2015. On Labeled Birooted Trees Languages: Algebras, Automata and Logic. *Information and Computation* 243 (2015), 222–248.
[12] D. Janin. 2016. A robust algebraic framework for high-level music programming. In *Second International Conference on Technologies for Music Notation and Representation (TENOR)*.
[13] M. Kambites. 2011. Free adequate semigroups. *Journal of the Australian Mathematical Society* 91 (2011), 365–390.
[14] M. V. Lawson. 1991. Semigroups and ordered categories. I. The reduced case. *Journal of Algebra* 141, 2 (1991), 422 – 462.
[15] M. V. Lawson. 1998. *Inverse Semigroups : The theory of partial symmetries*. World Scientific.
[16] J. Meakin. 2007. Groups and semigroups: connections and contrasts. In *Groups St Andrews 2005, Volume 2 (London Mathematical Society, Lecture Note Series 340)*. Cambridge University Press.
[17] K. S. S. Nambooripad. 1980. The natural partial order on a regular semigroup. *Proc. Edinburgh Math. Soc.* 23 (1980), 249âĂŞ260.
[18] A. van der Ploeg and K. Claessen. 2015. Practical principled FRP: forget the past, change the future, FRPNow!. In *Int. Conf. Functional Programing (ICFP)*. ACM, 302–314.
[19] B. A. Yorgey. 2012. Monoids: Theme and Variations (Functional Pearl). In *Proceedings of the 2012 Haskell Symposium*. ACM, 105–116.