Vitry manual

1 Introduction

Vitry is a functional programming language and an environment for representation and manipulation of music. The language has infix syntax and simple, expressive type system. The represented music can be output in a variety of formats, transcribed to musical notation or used as control data for sound synthesis.

The representation of music is abstract in the sense that it is not concerned with actual sound, but the formal structure of music. The basic idea is to give the user the tools needed to impliment any kind of *musical model*, while also providing implementations of standard cases.

Vitry may be used for composition, arranging, transcription or analysis. It is built for easy integration with other environments, particularly the following:

- Lilypond
- Sibelius
- SuperCollider

2 Getting started

Vitry targets the Java Virtual Machine, and may be used with almost any operating system. The only dependency is the Java runtime environment. To check if this is installed, open a shell and type:

```
$ java -version
```

If not, download a suitable implemementation. There are many alternatives.

2.1 Compiled distributions

Compiled distributions may be downloaded from the GitHub site: http://github.com/hanshoglund/vitry/downloads. If a version for your operating system is not available, download and compile a source distribution.

2.2 Source distributions

The build requirements are Git and Apache Ant. You may check if these are installed as follows:

```
$ git
$ ant
```

If not, download and install from the sites above or through your package management system.

As soon as the build requirements are in place, do:

```
$ git clone git@github.com:hanshoglund/vitry.git
$ cd vitry
$ ant
$ sudo ant install
```

You may append a directory to the install command. The default is `/usr/bin`.

2.3 Using the interpreter

The interpreter is typically the simplest way to interact with Vitry. To run it, simply type:

```
$ vitry
```

Vitry will print some setup information and enter a read-eval-print mode. In this mode, you may enter an expression followed by enter, and it will be evaluated and printed to you. Any expression may be entered. However, module declarations are not accepted.

TODO

The interpreter also accepts special commands commenced by a colon. These are not part of the Vitry language, but exist for ease of use in the interpreter. To see a list of available commands, use `:h` or `:help`.

```
  : help h        Prints help information.
  : quit q        Leaves Vitry.
```

TODO expand

3 The language

Vitry is similiar to other functional programming languages including Lisp and Haskell. This chapter will give a brief overview of functional programming but focuses on the concepts unique to Vitry[1].

3.1 Values

A functional language is mainly concerened with manipulating values. Broadly speaking, values are pieces of data that represent something. In Vitry, values are unique and non-changing. This is also true for compound structures, such as lists.

3.2 Expressions

Expressions are series of tokens each of which may produce a value. Expressions may be nested. Below are some examples of expressions along with their result[2].

```
1             => 1
2 + 3         => 5
(2 + 3) * 2   => 10
not true      => false
sin (pi/2)    => 1
```

The most common forms of expressions are outlined below. Detailed syntax and behaviour will be described in detail later on in this manual.

3.2.1 Literals

The simplest form of expression, used to create simple values like numbers and strings. Examples are:

```
1
22.5
foo
"Philippe"
```

[1] For good thorough introduction to functional programming, study a textbook such as *Structure and Interpretation of Computer Programs* by Abelson and Sussman

[2] The arrow-like sign not part of the language, but just a conventional way of wrinting what an expression evaluates to

3.2.2 Infix expressions

Consists of other expressions, along with operators. A familiar form is the arithmetic expressions. Examples are:

```
1 + 2
22 / 11
23 % 11
1, 2, 3, 4
true | false
```

3.2.3 Function calls

Consists of a callable expression followed by other expressions.

```
print "hello world"
not true
sum 1 2 3 4
```

3.2.4 Special forms

The so-called special forms are identified by the following keywords:

- `let` and `where` expressions used for binding variables.
- `fn` expressions, to define functions.
- `if`, and `match` expressions used for conditional evaluations.
- `do` expressions, used to carry out side-effects like input and output.
- `module` and `implicit` declarations.

3.3 Delimiters

Expressions may be nested using the following characters as delimiters:

```
() [] {}
```

TODO

3.4 Types

Types are used to group and reason about values. A type may be thought of as a common property of some values. Any value may be tested to see if it conforms to this property, if it does it is said to have the given type.

The type system of Vitry is dynamic in the sense that types are evaluated at runtime, and strong in the sense that arguments to functions are checked by default.

3.4.1 Booleans

The boolean type is written as `bool`. Its values are written as `true` and `false`.

3.4.2 Numbers

Vitry supports bignum natural, integer and rational numbers, as well as floating-point real and complex numbers. The types of these are written as `nat`, `int`, `rat`, `float` and `complex` respectively.

Natural, integers and rational numbers are written as sequences of digits. Vitry will automatically convert integers to rationals and vice versa:

```
152
42
-8
3/2
```

Floating point numbers may be written in several ways:

```
0.1
0.12e10
2e-5
0.5/2
```

We create a complex number by adding the suffix `i` to the imaginary part:

```
2i
10 + 1i
22.4 + 32e4i
```

Note that to get one imaginary unit you have to write `1i`, as `i` is not a number literal. Complex numbers in polar form may be entered using the `cis` function:

```
22 * cis 4
```

3.4.3 Strings

Strings are sequences of Unicode characters. The string type is written as `string`. String values are written inside double-quotes:

```
""
"test"
"\""
"\\"
"I hate music"
```

3.4.4 Atoms

TODO

3.4.5 Or types

TODO

Intersection types capture the notion of *inheritance* in object-oriented languages.

3.4.6 And types

TODO

Intersection types capture the notion of *composition* in object-oriented languages.

3.5 Bindings and scopes

TODO

3.6 Functions

TODO

3.7 Loops and recursion

TODO

3.8 Sequences

TODO

3.9 Predicates and matching

TODO

3.10 Side effects

TODO

3.11 Modules

TODO

3.12 Implicitness

TODO

4 Representating music

Vitry provides a large set of types and functions that simplifies the manipulation of musical data. These are defined in the language itself, giving the user of the power to define structures that operate on exactly the same level of abstraction as those provided with the language. Most of this chapter is devoted to descriptions of the standard model, but it shoul be clear that these conventional structures are by no means mandatory. On the contrary, the user is highly encouraged to provide alternate representations and the language is designed to faciliate that.

TODO music vs notation

4.1 Time

Perhaps the most general musical property, time turns out difficult to model in a simple yet coherent way. Thus we will provide several different, though related, time models and a simple taxonomy to keep track

of them, and their various properties[3].

We use a type `time` as the root of our hierarchy of time models. For simplicity, we will limit this type to linear and synchronous models, and use completely separate types when this is not the case. By *linear* we mean that `time` progresses consistently without repetition or jumps, and by *synchronous* that relations between `time` values can be taken to hold in all cases[4]. This will be sufficient to represent most conventional music. Nonlinear and nonsynchronous time will be covered in later sections.

TODO absolute and relative (time)

Time values are commonly used to represent *positions* in a bar-beat grid, as well as *durations*, meaning the difference between the onset and offset position of a certain event. There is no real need to make this distinction on the type level, but we will introduce two handy synonyms for `time` that can be used to prevent mix-ups when working larger time structures.

TODO make distinction between abs/rel *time scale* as above and abs/rel *notation* as in MIDI

```
type
  time         = absoluteTime | relativeTime
  absoluteTime = sec
  relativeTime = rat
  pos          = time
  dur          = time
  sec          = float
```

TODO tuples

TODO scaling tempo

TODO continous tempo scaling

TODO quantization

Relative time is simply represented as rational numbers, using the conventional note names as reference point. Thus the values `1, 1/2, 1/4, 1/4` may be read as whole note, half note, quarter note, quarter note.

For absolute time we use standardized units:

```
implicit type
  min  = 60 * sec
  hour = 60 * min
```

[3]Here time is taken to mean a measurable unit of time (as in the sentence “one second’s time”), not the amount of beats in a musical pulsation.

[4]As long as the conductor (or the transport system of the studio) behaves professionally.

```
  day  = 24 * hour
  Hz   = 1 / sec
  kHz  = hz * 10e3
  MHz  = hz * 10e6
```

TODO

4.2 Pitch

Pitch is readily represented as an absolute frequency value or as a postion in a scale.

TODO tuning systems TODO reference frequency

TODO

4.3 Events

Events is the abstract type representing discrete musical actions.

TODO

4.4 Processes

A musical process (not to be confused with a computational) is the abstract type representing continuous musical actions.

TODO

4.5 Instrumentation

The concept of instrumentation (or *orchestration*) may be generalized to represent the distribution of a set of events across a set of *performers* or *ensembles*. This concept is useful not only for orchestral music, but also for distribution of events across synthesizers, or even virtual performers such as sub-processes of a generative piece.

The atomic unit of instrumentation is the performer, defined as a receiver of musical events. There is no theoretical distinction between a vocal, instrumental or virtual performer, but we provide these synonyms for convenience.

```
type
  performer
```

```
singer      = performer
instrument  = performer
```

TODO choir, ensemble

TODO model conventional settings?

TODO arrangements, reductions

TODO

4.6 Spacialization

As instrumentation is concerned with distribution of events amongst discrete groups or individuals, we may also consider the distribution of events or processes in a spacial continuum.

Spacialization is generally mostly of interest in acousmatic music, as fine control of the spacial parameter is not available in the instrumental or vocal genres. However, if spacialization is thretened as continous distribution in an abstract sense, there are possibilities of using “virtual spaces” to control other musical parameters[5], or even model the real-life spacial setup of a group of instruments.

TODO

4.7 Nonlinearity

called `nonlinear` and `nonsync` TODO

4.8 Indeterminate structures

TODO

5 Miscellaneous topics

5.1 Calling foreign languages

TODO

[5]This would not have to be called spaces, but I find that name intuitive as the time dimension is already occupied

5.2 Setting up the environment

TODO

5.3 Replacing the syntax

TODO

5.4 Real-time

TODO

5.5 Networking

TODO

6 Reference

6.1 Syntax

```
expr
  : '(' inline ')'
  | '[' inline ']'
  | '{' inline '}'

  | 'fn' parameter* expr
  | 'let' binding* expr
  | 'do' '(' expr+ ')'
  | 'if' expr expr 'else'? expr

  | atom
  | natural
  | float
  | complex
  | string

parameter
  : atom ':' expr
```

```
binding
  : atom '=' expr

inline
    : apply (operator+ apply)+
    | apply
    | empty

apply
    : expr+
    | expr

empty
    :

operator
  Any of the following characters in any order
  ! # $ % & \ * + , - . / ; < = > ? @ \ ^ _ ` | ~ '

atom
  a-z or A-Z followed by any number of a-z, A-Z or 0-9

natural
  any number of 0-9

float
  TODO

complex
  TODO

string
  TODO
```

6.2 Dictionary

TODO

Value a

Token a

Expression a

Evaluation a

Type a

Function a

Predicate a

Sequence a