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 an expressive type system based on predicate matching. 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

1.1 About this manual

This manual is unfortunately very brief, and requires some knowledge of programming and music theory to be fully comprehended. Some more accesible tutorials should be added in the future.

2 First steps

2.1 Building and installing

Vitry targets the Java Virtual Machine, and may be used with almost any operating system. The only dependency is the Java runtime environment. The build requirements are Git and Apache Ant. You may check if these are installed as follows:

```
$ java
$ 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
```

Or if you are on Windows:

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

You may append a directory to the install command.

2.2 The interpreter

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

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:

```
:help
  Prints help information.

:info [obj]
  Displays information about the given value.

:compile [file]
  Compiles the given vitry file or directory, resulting in a class file.

:quit
  Leaves Vitry.
```

TODO

2.3 Scripts and compilation

While the interpreter provides a simple way to experiment and test out code, larger projects will require writing code in files. Vitry accepts text files or precompiled class files for execution. The only difference between the two is that class files may execute somewhat faster.

Vitry code is written in standard text files. Only the UTF–8 encoding is accepted. File names typically have the `.vitry` suffix, allthough this is not required. There is no difference between the kind of expressions allowed in the interpreter and in file source code except that the interpreter commands are not allowed in files. Vitry files may have a shebang line, allowing them to be executed in standard Unix shells.

In a large Vitry project, most source code files will be used to define modules and implicits. To make an executable program a module containin a main function is required.

```
in myApp
  main = do
    prompt "Please enter your name:"
    post + ("Hello " ++ _)
```

To excute a script, simply use:

```
$ vitry hello.vitry
```

3 The language

Vitry is a functional programming language. Well known languages of this type include Lisp, ML and Haskell. Like these languages, Vitry is centered around functions, values and expressions. Imperative operations are possible but not required except for basic input and output tasks.

The lexical conventions are very simple. There kinds of tokens are spaces, line breaks, keywords, operators and delimiters as well as literals for symbols, strings and numbers. Indentation levels are rewritten as delimiters before interpretation, allowing nested expressions to be written without a large amount of parentheses. Thus indentation may be ommited altogether in generated code. For details on the lexical sytax, see the final chapter of this manual.

3.1 Expressions

Functional languages are mainly concerened with the expression of values. Broadly speaking, values are pieces of data that represent something. Expressions are series of lexical tokens, each of which

produces a value. Below are some simple expressions along with their result.[1]

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

There are three basic forms of expressions, outlined below. Along with the special forms and delimiters, these make up the syntax of the Vitry language. The full syntax is given in the final chapter as well.

3.1.1 Literals

The simplest form of expression, literals are written representation of simple values such as numbers, strings or atoms. Examples are:

```
1
22.5
phillipe
"thomas"
```

3.1.2 Function application

Consists of an expression followed by one or more other expressions. The first expression is assumed to be a function, while the others are assumed to be its arguments.[2]

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

3.1.3 Infix expressions

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

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

[2] As in Haskell, functions may always partially applied. Thus function application may be seen as a left-associative binary operation.

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

3.1.4 Special forms

Special forms are identified special keywords, which are reserved for a particular use in the language:

- `let` and `where` for binding variables.
- `fn` for function definitions.
- `if` and `match` for conditional evaluation.
- `do` for carrying out side-effects like input and output.
- `in` and `imply` for module declarations
- `type` for type expressions.

The following operators are reserved for definitions, type restrictions and quotation respectively:

```
= : `
```

3.2 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 required to have types. In contrast to most languages, Vitry treats types as values as well. The type of types is a special value called `type`. Its type is called `type_` etc.

3.2.1 Booleans

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

3.2.2 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
```

3.2.3 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"
```

3.2.4 Symbols

Symbols are representations of unique values, such as `true` or `false`.

TODO binding usage

All the types described in this manual are in fact type values bound to symbols such as `bool`, `int`, `nat` etc.

3.2.5 Or types

TODO

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

3.2.6 And types

TODO

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

3.3 Delimiters

Delimiters consist of the characters `( ) [ ] { }`, and may be used to group expressions. Delimiters and must be written to balance. Thus the following expressions are all valid.

```
(1)
(1, 2)
[(1 + 2)]
{(1 + 2) * 3}
```

However, the following expression is not:

```
([1)]
```

Delimiters are commonly used to indicate precedence:

```
2 * 3 + 4   => 10
2 * (3 + 4) => 14
```

Delimiters may also be bound to functions. By default, standard parentheses `( )` do nothing, while brackets `[ ]` and braces `{ }` are bound to the functions `list` and `set` respectively. Thus, these delimiters may be thought of as literals for lists and sets, a fact which is acknowledged by their written representations.

```
[1, 2, 3] : list
  => true

{1, 2, 3} : set
  => true

[0 + 1, 1 + 1, 1 + 2]
  => [1, 2, 3]
```

TODO We need non-evaluating arguments to get expressions such as [(1, 2, 3)] to differ from [1, 2, 3].

3.3.1 Indentation

Vitry use indentation as a way of expressing nested expressions without actually having to write out all the delimiters. This is achieved by a process called indentation rewriting, which is performed on all code before interpretation.

TODO

```
john paul
george ringo
  =>
(john paul)
(george ringo)

john paul
  george ringo
  =>
(john paul
  (george ringo))

john
  paul
    george ringo
  =>
```

```
(john
  (paul
    (george ringo)))

john
  paul
  george
    ringo
  =>
(john
  (paul)
  (george
    (ringo)))
```

3.4 Bindings and scope

Local bindings may be used with the `let` or `where` forms. They are identical except that the let form expects the definition before the scoped expression and where the reverse. Bound expression evaluates to the value of the scoped expression with the given values bound in.

```
let atom = expr atom = expr ... expr

expr where atom = expr atom = expr ...
```

Example :

```
let foo = 1
    bar = 2
  foo + bar
=> 3

foo + bar where
  foo = 1
  bar = 2
=> 3
```

Bindings are resolved by lexical scoping, thus inner bindings always override outer:

```
let foo = 1
```

```
  let foo = 2
    foo
=> 2
```

3.5 Functions

TODO

At the semantic level, Vitry makes no distinction between functions, delimiters and operators.

3.6 Loops and recursion

TODO

3.7 Sequences

TODO

3.8 Predicates and matching

A *predicate* is a function on the form `? -> bool`.

TODO

3.9 Side effects

TODO

3.10 Modules

TODO

3.11 Implicitness

TODO

4 Musical representation

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.

4.1.1 Time scale

TODO absolute and relative (time)

4.1.2 Positions and durations

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 *durations* as in MIDI

[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 properly.

```
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
  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.

4.2.1 Scales

TODO tuning systems

4.2.2 Tuning

TODO reference frequency

TODO

4.3 Events

Events is the abstract type representing discrete musical actions. TODO

4.3.1 Notes

TODO

4.3.2 Rests

TODO

4.3.3 Phrasing

TODO

4.3.4 Tags

TODO

4.4 Processes

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

TODO

4.4.1 Dynamics

TODO

4.4.2 Glissandi

TODO

4.4.3 Tremolo and iterations

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 treated a continous distribution in an abstract sense, there are possibilities of using “virtual spaces” to control other musical

parameters.[5] It may also be used in conjunction with instrumentation to model the real-life spacial setup of an ensemble.

TODO

4.7 Nonlinearity

called `nonlinear` and `nonsync` TODO

4.8 Indeterminate structures

TODO

5 Transcription and performance

6 Miscellaneous topics

6.1 Calling foreign languages

TODO

6.2 Setting up the environment

TODO

6.3 Replacing the syntax

TODO

6.4 Real-time

TODO

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

6.5 Networking

TODO

7 Reference

7.1 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
```