Vitry Manual

1 Introduction

Vitry is a functional programming language and an environment for representation and manipulation of music. The language features a succinct, expressive syntax and a powerful 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 notions of time, pitch, phrasing etc.

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 The language

Like most functional languages, Vitry depends on functions, values and types are the principal units of abstraction. Mutable state is avoided and recursion and looping are used interchangebly.

Unlike most languages, Vitry treats both functions and types as first class values, that may be referenced and passed to functions like any other. Evaluation is strict by default, but lazy evaluation is possible and lists are always lazy. The type system is based on the notions of implicit conversions.

TODO explain structural equivalence

2.1 Lexical conventions

The lexical conventions are very simple. All code is parsed into one of the following kinds of tokens by the lexer:

- Spaces
- Line breaks
- Operators and delimiters
- Literals for symbols, strings and numbers
- Keywords for special forms

Indentation levels are *expanded* to delimiters before interpretation, allowing nested expressions to be written without a large amount of parentheses. Thus indentation may be ommited altogether if delimiters are used instead. For details on the lexical sytax, see the final chapter of this manual.

2.2 Types

TODO (determine) explain type equivalence

2.2.1 Booleans

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

2.2.2 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
```

2.2.3 Strings

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

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

2.2.4 Symbols

Symbols are representations of unique values. As values equivalence is

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

The quote character may be used to access operators and delimiters as symbols. This is used to assign functions to operators and delimiters:

```
`++ = concat
`[] = list
`{} = set
```

2.2.5 Or types

TODO

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

2.2.6 And types

TODO

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

2.3 Expressions

Expressions are series of tokens generated by the lexer. Each expression produce a single value. We use the characters => to indicate evaluation:

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

There are five basic forms of expressions, namely literal, applicative, infix and special form. Along with the delimiters, these make up the syntax of the language. A formal specification of the syntax is given in the final chapter.

2.3.1 Literal expressions

The kind of expression, literals are written representation of simple values:

```
1
22.5
guillaume
"thomas"
```

The `read` and `show` functions may be used to transform such values into their written representation and vice versa. The show function is used by the interpreter to display values.

```
read "22.3"         => 22.3
read "josquin"      => josquin
read "\"josquin\"" => "josquin"

show josquin        => "josquin"
show "josquin"      => "josquin"
```

2.3.2 Application

This form 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.[1]

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

2.3.3 Infix expressions

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

```
1 + 2
23 % 11
1, 2, 3
-1
~true
true | false
```

Operators are made up of distinguished operator characters. Prefix and infix operators are allowed, but not postfix. The following operators are reserved for definitions, type restrictions and quotation respectively:

```
= : `
```

[1]As in Haskell, functions may always partially applied. Thus function application may be seen as a left-associative binary operation, which is true to the original concept of the lambda calculus.

2.3.4 Special forms

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

- `let` and `where` for binding variables.
- `loop` and `recur` for traversal.
- `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.

2.3.5 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 used to indicate precedence:

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

Delimiters may also be bound to functions, to which the enclosed expression will be applied. Thus look exactly like operator binding:

```
`() = fn x:nat x
`[] = fn x:nat x + 1
`{} = fn x:nat x * 2
```

By default, standard parentheses `()` are bound to the `id` function (so they do nothing).

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 the `show` method.

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

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

repeat 0 2
  => [0, 0]
```

2.3.6 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)))

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

2.3.7 Comments

TODO

2.4 Functions

Functions are defined by the `fn` special form:

```
fn expr
fn parameter : type expr
fn parameter : type parameter : type ... expr
```

TODO

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

2.5 Bindings

Bindings is the notion of assigning references (values) to symbols. There are three kinds of bindings: global, parametric and local.

Global bindings are typically used to define functions and types and can be accessed from any other expression. Despite the name, they are typically encapsulated into modules. Global bindings can be used declarative (i.e. without any need to concern oneself about evaluation order):

```
b = a + 2
a = 2

c = d
```

```
d = c
type c, d
```

Parametric binding is the process of substituting a the parameters of a function with the given arguments upon evaluation. Local bindings are "one-off" associations that apply to a single expression. The only difference between local and parametric binding except that local binding is performed directly upon evaluation of the given expression. Both are resolved though lexical scoping, thus inner bindings always override outer:

```
let foo = 1
  let foo = 2
    foo
=> 2

(fn [foo bar]
  fn [foo bar]
    bar foo
  bar foo) 1 2
=> 1 2
```

2.5.1 Let and where

The `let` or `where` forms provide local binding. They are identical except that the let form expects the bound expression first, and the where form last. Bound expression evaluates to the value of the scoped expression with the given values bound in.

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

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

Example :

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

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

The `let` and `where` forms are very useful for creating local variables or factoring out expressions to make them more readable.

2.5.2 Loop and recur

The `loop` and `recur` forms (borrowed from Clojure) are the most efficient way to traverse data structures.

TODO

```
loop atom = expr expr
loop atom = expr atom = expr ... expr

recur expr
recur expr expr ...
```

2.5.3 Do

TODO

2.6 Conditions

2.6.1 If

2.6.2 Match

2.7 Modules

2.7.1 Import

2.7.2 Function syntax

2.7.3 Type syntax

2.7.4 Implicits

3 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 the built-in structures.

TODO music vs notation

3.1 Time

Musical time [2] can be represented in numerous ways. We use a type `time` as a general representation of linear and synchronous time. 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. This will be sufficient to represent most conventional music. Nonlinear and nonsynchronous time will be covered in later sections.

3.1.1 Time scale

TODO absolute and relative (time)

[2] Here 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.

3.1.2 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

```
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

3.2 Pitch

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

3.2.1 Scales

TODO tuning systems

3.2.2 Tuning

TODO reference frequency

TODO

3.3 Events

The `event` is the abstract type representing discrete musical actions. TODO

3.3.1 Notes

TODO

3.3.2 Rests

TODO

3.3.3 Tags

TODO

3.4 Phrasing

TODO

3.5 Processes

The `process` is the abstract type representing continuous musical actions.

TODO

3.5.1 Dynamics

TODO

3.5.2 Glissandi

TODO

3.5.3 Tremolo and iterations

TODO

3.6 Instrumentation

An `instrumentation` represents the distribution of a set of events across a set of *performers* or *ensembles*. This is useful not only for instrumental 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
```

3.6.1 Performers

3.6.2 Choirs

3.6.3 Ensembles

3.6.4 Standard setups

3.6.5 Arrangements

3.6.6 Reductions

TODO

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

TODO

3.8 Nonlinear structures

TODO

4 Transcriptions

4.1 Standard notation

4.1.1 Metrical grouping

4.1.2 Spelling

4.1.3 Style options

4.1.4 Formats

4.2 Free-form notation

4.3 MIDI

5 Practical topics

5.1 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.

5.2 The interpreter

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

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.

5.3 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
```

5.4 Setting up the environment

TODO

5.5 Calling foreign languages

TODO

5.6 Replacing the syntax

TODO

5.7 Real-time

TODO

5.8 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
```