Vitry manual

Preface

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 representation of music is abstract in the sense that it is not concerned with actual sound, but the formal structure of music. These representations can be transcribed to standard musical notation or used as control data for sound synthesis.

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. 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.

Precompiled distributions of Vitry may be downloaded from here. If an installer is not available for your operating system, you have to download and compile a source distribution.

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
$ ant install /usr/bin
```

You may of course substitute any directory you want for `/usr/bin`.

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.

2. The language

Vitry is similiar to other functional programming languages. This chapter will give a brief overview of the concepts used in Vitry[1].

Basics

Any functional language is first and foremost concerened with manipulating values. Values are pieces of data that represent something. In Vitry, all values are unique and non-changing.

Expressions are series of tokens each of which may produce a value. Expressions may be nested using parentheses. Thus any program could be thought of as a single expression. There are various ways of writing expressions, of which the most common can be seen in this example:

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

[1]For a complete description of functional programming concepts, see Structure and Interpretation of Computer Programs by Abelson and Sussman.

Types

Types are used to group and reason about values in a logical way. This help us think about the values we are manipulating and prevent us from doing mistakes. 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.

Booleans

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

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
```

Strings

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

```
"I hate music", "But I love to sing"
```

Atoms

TODO

Or types

TODO

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

And types

TODO

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

Bindings and let

TODO

Functions

TODO

Loop and recur

TODO

Sequences

TODO

Effects and do

TODO

Modules

TODO

Implicitness

TODO

3. Representing music

Vitry defines a musical model through a set of types and functions. All musical structures are defined in the language itself. This gives the user of the power to define custom data structures that operate on the same level of abstraction as those provided with the language. The core musical model is agnostic to musical style or writing system, instead it simply represent music as a set of events.

clear distinction between the concepts of *music* and *notation*, much like the separation of content and presentation commonly used in text processing. By using such a separation, music can be defined and manipulated as a kind of algebraic structure, apart from notational conventions. Another benefit of this strategy is that the same music could easily be translated to any kind representation.

The process of converting a musical value to a notation is called *transcription*, while the reverse operation is called *interpretation*. These operations will be explained more fully in the next chapter.

Token a

Expression a

Evaluation a

Type a

Function a

Predicate a

Sequence a