Faustine: a Vector Faust Interpreter Test Bed for Multimedia Signal Processing

Karim Barkati, Haisheng Wang, Pierre Jouvelot

Faustine: a Vector Faust Interpreter Test Bed for Multimedia Signal Processing

System Description

Karim Barkati, Haisheng Wang, and Pierre Jouvelot
name.surname@mines-paristech.fr

MINES ParisTech, France

Abstract. Faustine is the first interpreter for the digital audio signal processing language Faust and its vector extension. This domain-specific language for sample-based audio is highly expressive and can be efficiently compiled. Faustine has been designed and implemented, in OCaml, to validate the Faust multirate vector extension proposed in the literature, without having to modify the sophisticated Faust scalar compiler. Moving to frame-based algorithms such as FFT is of paramount importance in the audio field and, more broadly, in the multimedia signal processing domain. Via the actual implementation of multidimensional FFT and morphological image processing operations, Faustine, although unable to process data in real time, illustrates the possible advantages and shortcomings of this vector extension as a language design proposal. More generally, our paper provides a new use case for the vision of interpreters as lightweight software platforms within which language design and implementation issues can be easily assessed without incurring the high costs of modifying large compiler platforms.

1 Introduction

Domain-specific languages (DSLs) are high-level, specialized, abstract programming languages that help shrink the "semantic gap" between the concepts of a particular application area and the program implementation level. These languages are, by essence, more often upgraded or extended than traditional programming languages, since the unavoidable changes to the underlying business logic often call for the introduction of new traits in the corresponding DSLs [10]. This makes the design language phase – where one looks for defining a proper balance between the choice of programming features, their practical relevance and their performance cost – an almost constant endeavor.

Finding an appropriate set of programming language features calls thus for a trial-and-error design and implementation approach, which may end up being a costly proposition when the corresponding language evaluation platform must be continuously tweaked to test new proposals. Reaching an acceptable language design is even more complicated when one deals with advanced languages that target compute-intensive applications such as signal processing. Indeed, such

performance requirements are usually only met by sophisticated compilation systems that incorporate advanced optimization phases, making such software platforms unwieldy and difficult to adapt on-the-fly to test new language ideas.

Language interpreters are often considered as simple educational tools to introduce semantic language concepts (see for instance [22]), illustrate one of the Gang-of-Four design patterns (Interpreter Pattern) [9] or motivate the introduction of compilers for performance purposes. Yet, interpreters can also be invaluable tools to assess the adequacy of various programming language traits (see for instance the seminal paper [26]), and thus help address the cost-vs-design conundrum that plagues DSLs. We illustrate this idea with Faustine, an OCaml-based interpreter for the Faust programming language. Faustine implements the core of Faust, a functional DSL dedicated to advanced audio signal processing, and is thus a useful tool in itself, for instance to debug Faust programs that use new constructs. But the development of Faustine is also motivated by the desire of testing the "interpreters as DSL design-assistant tools" idea outlined above (see also [5], an interesting blog on other positive aspects of interpreters for traditional languages).

As a case study, we augmented Faust with the vector extension proposal introduced in *Dependent vector types for data structuring in multirate Faust* [17] and tested its practical applicability using a set of typical benchmarks. As explained in Section 6, these experiments showed us that some unanticipated problems lurked within the current vector design; discovering such issues early on is what the Faustine prototype is all about. On the contrary, a positive byproduct of the introduction of vectors within the Faust programming paradigm is that such an extension not only opens the door to important audio analysis techniques such as spectral algorithms to the Faust community, but may even extend the very domain Faust has been designed for. Indeed we show how key image processing operations can be expressed in Faust, and run in Faustine.

To summarize, the major contributions introduced in this paper are:

- an illustration of the "interpreters as language design-assistant tools" idea, which, even though not new, we think could be looked at in a new light when dealing with DSLs and their unusual life cycle and performance requirements. In our case, this approach proved also quite cost-effective, since we were able to assess our design ideas after the two months it only took to implement Faustine, a short time given we were not very knowledgeable about OCaml at first;
- Faustine[1], an OCaml-based interpreter for the functional audio signal processing language Faust;
- the first implementation of the Faust vector extension proposed in [17] within Faustine, seen as a test bed for assessing the adequacy of new language constructs for Faust;
- a first look at the wider applicability of the Vector Faust programming model in the more general setting of multimedia signal processing, providing some insights to its possible use for image processing.

[1] http://www.cri.mines-paristech.fr/∼pj/faustine-1.0.zip

After this introduction, Section 2 introduces the Faust project and its core language. Section 3 outlines the vector extension proposal, our main motivation for the development of Faustine. Section 4 describes the main features of Faustine, our Faust interpreter. Section 5 provides first experimental evidence of the practicality of its use, including running examples illustrating the applicability of the Faust vector extension to key applications such as Fast Fourier Transform (FFT) and image signal processing. Section 6 highlights some of the key preliminary results coming from the use of the Faustine system, yielding some ideas about future work. We conclude in Section 7.

2 Faust

Faust[2] (Functional Audio Stream) is a DSL originally designed to help implement real-time audio applications. Designed at Grame, a French center for music creation located in Lyon, France, the Faust project as a whole has been under development since 2002. It has been instrumental in the development of many audio applications, such as the open-source Guitarix[3] and commercial moForte[4] guitar synthesizers. It has also been an enabling technology in many of the music works and performances created at Grame. The Faust language is taught in many music-oriented institutions over the world, such as Stanford University's CCRMA and French IRCAM and Université de Saint-Etienne.

2.1 Design

DSLs derive from the knowledge of a particular application domain. This generally puts constraints upon the kind of programming constructs they provide. One popular approach is to embed such knowledge within an existing programming language [15]; this provides a general framework within which applications can be programmed. For the domain of audio processing addressed by Faust, such a generality was not deemed necessary. Deciding not to go for an embedded DSL in turn opens the door to specific optimization opportunities, which might be unreachable for more general programming languages. This is particularly true for run-time performance.

Given the high computational load required by audio signal processing programs [2], one of the requirements for Faust has indeed been, from the start, to strive to reach C++-like run-time performance, while providing high-level constructs that appeal intuitively to audio-oriented engineers and artists [20, 21, 3]. To reach such goals, the design of Faust adopts an unusual approach, being structured in two completely different parts:

- the Faust *core* language is a functional expression language built upon a very limited set of constructs that define an algebra over signal processors, which

[2] http://faust.grame.fr
[3] http://sourceforge.net/projects/guitarix
[4] http://www.moforte.com

are operators taking (theoretically infinite) streams of samples as input and yielding streams of processed samples as output;
- the Faust *macro* language is a version of the full-fledged, untyped lambda-calculus, that can be used to define parametrized functions and allows currification.

Faust expressions can mix freely constructs taken from these two language components. Full-fledged Faust programs are defined as sets of identifier definitions `i = e;` or macro definitions `f(x,y,...) = e;`. All uses of an identifier or a macro, e.g., `f(3,s,...)`, are expanded at compile time to yield only core expressions. The identifier `process` denotes the main signal processor.

2.2 Core Faust

Faust core expressions are signal processors, inputting and outputting signals. Even a constant such as `1` is seen as a signal processor that takes no input signals, and outputs one signal, all samples of which have value 1. Three main constructs, similar to the combinators used in a language such as Yampa[5], a DSL built on top of Haskell, are operators of what amounts to an algebra of signal processors.

- The "`:`" sequential combinator pipes the output of its first argument into the input of its second. Thus "`1 : sin`" is a core expression that takes no input (as does `1`) and outputs a signal of samples, all of value `0.841471`.
- The Faust parallel combinator is "`,`" and is the dual of "`:`". Here, "`1,2 : +`" takes again no inputs, but pipes two constant signals into the "`+`" signal processor, yielding a stream of samples, all of value 3.
- The Faust recursion combinator is "`~`". The output samples of its second argument are fed back, after a one sample delay, into the inputs of its first argument.

For instance, the signal that contains the infinite sequence of successive integers is defined by "`_ ~ (1,_ : +)`", where "`_`" is the identity signal processor, in Faust. Figure 1 provides a graphic explanation of the inner working of the "`~`" construct; the displayed diagram corresponds to the Faust program `process = _ ~ (1,_ : +);`. Note that the small square on the back edge denotes a 1-sample delay, and that all signals are initialized by default to 0.

The last two main constructs of the signal processor algebra are the fan-out "`<:`" and fan-in "`:>`" combinators. A fan-out duplicates the output signals of its first argument to feed the (supposedly more numerous) input signals of its second argument. A fan-in performs the dual operation, combining signals that end up mixing in the same input signal with an implicit "`+`" operation.

[5] http://www.haskell.org/haskellwiki/Yampa

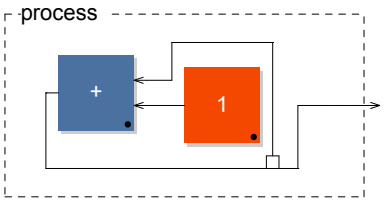

Fig. 1. The infinite stream of successive integers. The Faust platform provides a SVG block-diagram prettyprinter for Faust (text) definitions

2.3 Implementation

Faust is a purely functional specification language, operating at the audio sample level. The Faust compiler[6] makes such specifications executable. Since all macro-level constructs are expanded-away before code generation, the compiler can devote all its attention on performing efficient code generation at the Core Faust expression level. Performance being key, this highly optimized sequential compiler uses C++ as its target language. Its more than 150,000 lines of C++ code implement a wealth of optimization techniques, needed to enable real-time processing of computing-intensive audio applications. When even more performance is needed, parallel code, using OpenMP pragmas, can be generated.

In addition to the compiler itself, the Faust software suite offers a graphical IDE (FaustWorks) and many architecture files enabling its use via standard audio interfaces or plugins such as VST, Jack, ALSA, iOS and Android. Some important computer music environments such as Max/MSP, CSound or OpenMusic embed a standalone version of the Faust compiler, opening up the way to the use of Faust within foreign systems. A SaaS-version of the Faust compiler is available on the Faust web site.

3 Faust Vector Extension

Faust current design, focused on audio signal processing, assumes that all signals carry scalar floating-point or integer values. Yet, many digital signal processing (DSP) operations such as FFT and its derivatives operate on finite arrays of values, or frames. Such a feature is lacking, and even more so if one envisions to extend Faust application domain to others, such as image processing. A proposal for a simple vector extension has been introduced [17], which we briefly summarize below.

Faust is a typed language: a signal processor is typed with the type of the scalars carried over its signal input and output arguments. To ensure efficient

[6] http://sourceforge.net/projects/faudiostream

compilation, these types are dependent, in that each type includes an interval of values; e.g., a signal of type $\mathsf{float}[0;10]$ can only carry floating-point values within the interval [0;10]. The proposed vector extension builds upon this typing mechanism by adding a *rate* information to types; the rate or frequency f of a signal is the number of samples per second this signal is operating at. This rate information is, in turn, modified when dealing with vector operations. In short, a `vectorize` construct takes an input signal of rate f and a fixed size signal n and generates a signal of vectors of size n, at a rate f/n. The dual operation, `serialize`, takes a signal at rate f of vectors of size n, and outputs the serialized vector components in its output signal, at rate $f \times n$. Note that this scheme imposes that array sizes are known at compile time; what might appear as an unacceptable constraint is in fact quite handy within Faust, with its two-level design approach and its emphasis on efficiency for audio applications.

If `vectorize` and `serialize` are the constructor and destructor of the vector algebra, component-wise operations are still needed. The current proposal, in tune with the minimalism of Faust design, offers only two constructs, called *pick* (noted by "`[]`") and *concat* ("`#`"). To define these operations, while also providing a flavor of Faust extended typing system and how it closely constraints Faust expression construction, we give below their typing schemes:

- `#` : $(\mathsf{vector}_m(\tau)^f, \mathsf{vector}_n(\tau)^f) \rightarrow (\mathsf{vector}_{m+n}(\tau)^f)$;
- `[]` : $(\mathsf{vector}_n(\tau)^f, \mathsf{int}[0; n-1]^f) \rightarrow (\tau^f)$.

All italic variables are supposed to be abstracted, to form type schemes. Concatenating two input signals carrying vectors of size m and n is possible only if they have the same rate, here f: the concatenated output signal operates at the same rate, but carries values that are vectors of size $m+n$, formed, at each time tick, by the concatenation of the two corresponding vectors in the input signals. Dependent typing shines in the case of pick operations: there, given vector values v carried by an input signal at rate f and a signal of scalar indexes i, which have to be integers within the bounds $[0; n-1]$ of the input vectors, pick creates an output signal, at rate f, formed with the components v_i, at each time tick.

To illustrate how the vector datatype can be used, we provide in Listing 1.1 a n-fold subsampling signal processor `subsampling(n)`. Running a subsampling-by-2 process over the list of successive integers (refer to Figure 1, if need be) yields a signal of successive odd integers.

Listing 1.1. n-fold subsampling signal processor

```
subsampling(n) = (_,n) : vectorize : [0];

integers = _ ~ (1,_ : +);
process = integers : subsampling(2);
```

4 Faustine

In this section, we motivate our decision to design and implement Faustine, while highlighting some of its salient features.

4.1 Motivation

We intend to ultimately extend the current Faust compiler with the vector API introduced above. Yet, adding this capability to the many tens of thousands of lines of C++ code of such a large program is a major undertaking. Moreover, it seems unwise to commit to a full-fledged implementation without validating our extension proposal in the first place. Thus, implementing a lightweight interpreter such as Faustine appears as a simple way, in addition to the intrinsic value of such a system for testing and debugging purposes, to provide a test bed for checking the validity and practical applicability of the proposed vector extension on actual examples.

The interpreter route is even more appealing given the nature of the Faust language we emphasized at the beginning of Section 2. Indeed, Faustine has largely benefited from its two-tiered structure, core and macro. Faust macro constructs are first processed out by the original Faust compiler, which only had to be slightly adapted, at the parser and SVG generator levels, to handle the few idiosyncratic syntactic features of the vector extension. The resulting expression is then fed to Faustine, which has been designed to only address core language expressions.

Note that an interpreter can sometimes even be converted into a full-fledged compiler using semi-automated techniques such as partial evaluation, as shown for instance in [13], which interestingly is also looking at DSLs for signal processing. Yet, in our case, we intend to eventually provide our vector extension as an upgrade to the existing Faust compiler. One reason is that we would like to leverage the wealth of optimization techniques that already exist in the current compilation infrastructure. Another one is that the modifications required to handle vectors efficiently, something our users would want, are going to be tricky. Indeed, Faust operations work at the audio sample level, and each of these samples is currently a scalar. Dealing with vector-valued samples and their corresponding more complex data structures (Reference [17] even suggests to introduce records) is going to require significant design thinking to handle memory management issues in an efficient manner.

4.2 OCaml for Executable Specifications

Faustine is an interpreter designed for testing purposes, and not for operational usage. As such, a high-level implementation language is called for, since rapid specification modification cycles can be excepted, for which a flexible and expressive programming paradigm and environment are of paramount importance [16]. Since performance is not the primary concern here, one must keep an eye on this issue when dealing with real life examples. Even though there exist frameworks such as K [7] that can be used to automatically derive interpreters, we chose OCaml for a couple of reasons.

– First, its mix of functional and object-oriented paradigms enables close-to-specification implementations. Indeed, one can even view OCaml as a language within which to express executable specifications [18, 23], in particular

when using denotational-style definitions, as is the case in the vector extension paper on which Faustine is based [17].
- OCaml sports a wide variety of libraries, and in particular a binding to the `libsndfile` package[7]. This C library handles I/O operations over audio files encoded in the WAV or AIFF formats, and comes in handy when performing audio processing applications.
- In addition to the functional and OO paradigms, OCaml offers imperative constructs, which are useful, when handled with care, for performing certain optimizations such as memoization.

4.3 Implementation

Faustine is an off-line interpreter; in particular, no interactive evaluation loop is provided. Instead, Faustine takes a Faust program file (.dsp) and evaluates it, taking as input the standard input file and generating processed data on standard out. First, the original Faust compiler is called to preprocess the .dsp file, in order to eliminate all macro calls and generate a Core Faust expression. This one is passed to Faustine, which parses it and evaluates it sequentially according to the semantics defined in [17]. Input and output signal data can be encoded in two possible formats: WAV and CSV (ASCII values separated by commas), the latter being useful for spectra (see Subsection 5.1) and images.

Following Faust semantics, every expression in a program is considered as a Faust signal processor. In turn, each processor consists of subprocessors connected via Core Faust constructors by signals. In Faustine, signals are defined as OCaml functions of time to values; here "time" represents the succession of clock ticks `t`, implemented as integers. More specifically, one has:

```
class signal : rate -> (time -> value) -> signal =
  fun freq_init -> fun func_init ->
    object (self)
      method frequency = freq_init       (* signal rate *)
      method at = func_init              (* signal initial value *)
      method add : signal -> signal
      ...
      method vectorize : signal -> signal
      method serialize : signal
```

As shown above, a signal sample rate is a key property defining a signal in our multirate context. A different sample rate is computed when a given signal is vectorized or serialized, e.g., as in the following:

```
method vectorize : signal -> signal =
  fun s_size ->
    let size = (s_size#at 0)#to_int in
```

[7] *Libsndfile* is a cross-platform API for reading and writing a large number of file formats containing sampled sound (http://www.mega-nerd.com/libsndfile).

```
if size <= 0 then
  raise (Signal_operation "Vectorize: size <= 0.")
else
  let freq = self#frequency#div size in
  let func : time -> value =
    fun t ->
      let vec = fun i -> (self#at (size * t + i))#get in
      new value (Vec (new vector size vec)) in
  new signal freq func
```

The main job of Faustine is to construct the dynamic relationship between the input and output signals of a processor. When executing a Faust program, all subprocessors are synchronized by a global clock. Every time the clock ticks, subprocessors pull the current value of their incoming processors' signals, and refresh the values of their output signals. For most non-delay processors, output signals only depend upon the current value of the input signals. Delay modules, like the primitive "mem" that uses a one-slot memory, depend on previous input frames; Faustine uses arrays to memoize signal values to avoid computing values more than once.

Faustine deals with all but GUI Faust constructs in only 2,700 lines of code, a mere 100 of which are dedicated to the vector extension design we were interested in evaluating. It took about 2 months to implement, even though we were not very knowledgeable about OCaml at the start; so, presumably, seasoned programmers could have completed this task in a shorter amount of time. Yet, this enabled us to assess in a matter of days the issues regarding the Faust vector extension and its shortcomings (see Section 6).

5 Experiments

Faustine is able to handle many Faust programs, although its off-line nature prohibits the use of GUI elements. This does not limit its usability in our case, since we are mostly interested in proof-of-design issues. To illustrate the expressive power of our Faust vector extension and the possible generalization of its application domain to non-audio multimedia systems, we provide below two significant examples. This first one is the implementation of FFT and the second, an image processing application, LicensePlate, based on mathematical morphological operators.

5.1 FFT

The Discrete Fourier Transform (DFT) of an N-element real-valued vector x_n, in our case a frame of N successive signal values, is an N-element vector X_k of complex values. This vector can informally be seen as denoting a sum of amplified sine and cosine functions. The frequency-to-amplitude mapping of these functions is called the signal *spectrum*. In practice, one has:

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-2\pi k n i}{N}}, k = 0, ..., N-1.$$

The Fast Fourier Transform (FFT) is an efficient algorithm that uses recursion to perform a DFT in $O(N \log(N))$ steps in lieu of $O(N^2)$:

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} e^{\frac{-2\pi m k i}{N/2}} + e^{\frac{-2\pi k i}{N}} \sum_{m=0}^{N/2-1} x_{2m+1} e^{\frac{-2\pi m k i}{N/2}}.$$

In the FFT process shown in Listing 1.2 (part of File fft.dsp), an input stream of real scalars is first vectorized in vectors of size 8. The eight elements of the vector are accessed in parallel through `picks(8)`, and then converted to 8 complex numbers in parallel by `real2pcplx(8)`. We implemented the `complex.lib` complex algebra library, a complex number being implemented, in its polar and Cartesian representations, as a 2-component vector.

Listing 1.2. Faust 8-sample length FFT (excerpts)

```
import ("complex.lib");

picks(n) = par(i, n, [i]);
fft(n) = _ <: picks(n) : real2pcplx(n) : shuffle(n) : butterflies(n);
process = vectorize(8) : fft(8) : pcplx_modules(8) : nconcat(8) : serialize;
```

The 8 complex elements are then reshuffled, and fed to a so-called butterfly processor (see Figure 2). The output of the recursively-defined butterflies (omitted here) are complex numbers. Their moduli are kept, and represent the spectrum. The eight real scalars in parallel are repacked into a vector of size 8 by `nconcat(8)`, and then serialized to produce the output stream, which represents the spectrum.

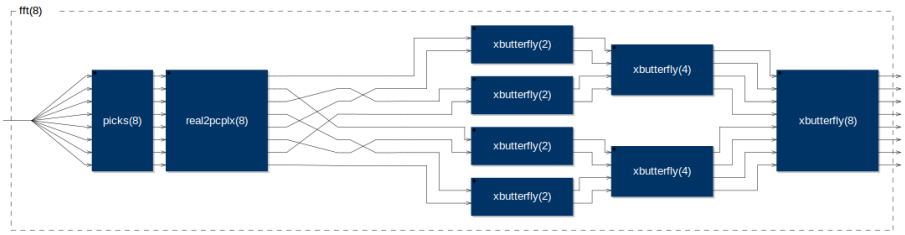

Fig. 2. FFT shuffling and butterfly

We ran a small experiment to illustrate the style of FFT outputs generated via Faustine. We fed fft.dsp the signal generated by the sum of four audio sine

waves in WAV format (1.378 kHz, 2.067 kHz, 16.536 kHz and 22 kHz, sampled at 44.1 kHz) as given in Listing 1.3, where `s(f)` denotes a sine wave function at Frequency `f`, and `t` is the list of successive integers, starting at `0`. The output of `process` was encoded as a .csv file, and is here plotted in Figure 3, using Octave[8].

Listing 1.3. Sum of 4 sine waves

```
import("math.lib");
samplerate = 44100;

process = s(1378) + s(2067) , s(16536) + s(22000) : + : /(4) ;
s(f) = 2.0*PI*f*t/samplerate : sin;
t = (+(1) ~ _) - 1;
```

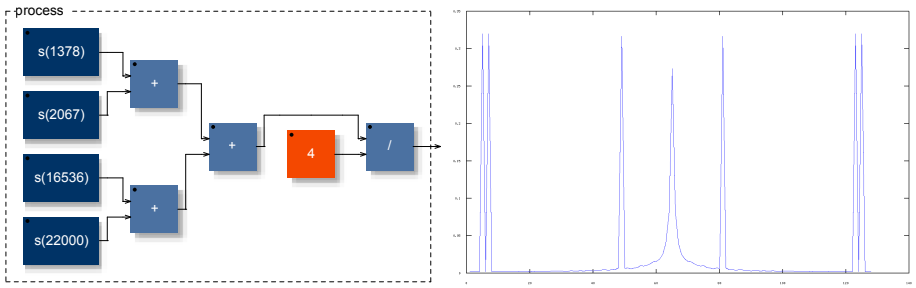

Fig. 3. FFT spectrum output of 4 sine waves: sum generation block diagram (left) and resulting analysis output (right)

5.2 Image Processing

The audio processing origins of Faust do not, a priori, preclude its usage in other domains. This should be even more the case once equipped with the vector extension described above. To test this hypothesis, we looked at how some image processing applications could be implemented in Vector Faust. As a case study, we chose LicensePlate, a car plate identification algorithm based on mathematical morphology operations.

Mathematical morphology [25] is a broad set of image processing methods based on shapes. The basic idea is to probe an image with a simple pre-defined shape, seen as a structuring element. The value of each pixel in the output image is determined by a comparison between the corresponding pixels in the

[8] Recall that, for real numbers x_n, the complex numbers X_{N-k} and X_k are conjugates, and have thus the same modulus.

input image with its neighbors, defined by the structuring element. Dilation is an important operation in mathematical morphology that uses this approach: the value of the output pixel is the maximum value of all the pixels in the input pixel's neighborhood (see Figure 4).

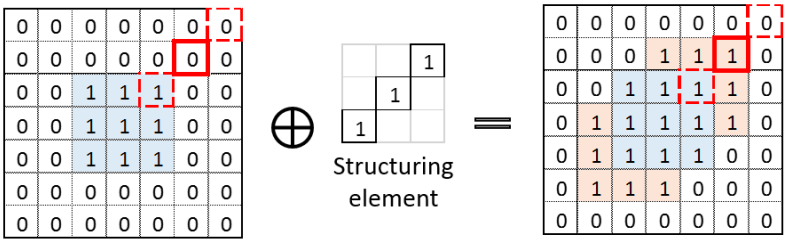

Fig. 4. Dilation $A \oplus B$ of a binary image A by a 3-point structuring element B

Implementing morphological operations in Vector Faust requires examining a 2D neighborhood of each pixel. A general solution is to examine one pixel in the neighborhood at a time, and then combine all the output images. Moreover, one can show that the image dilated by any pixel in the structuring element can be dilated firstly by line, then by column, using the associativity of the dilation operation $\oplus$ (see Figure 5).

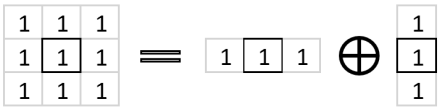

Fig. 5. Decomposition of a 3x3 square structuring element

For the example of Figure 4, one can use the code in Listing 1.4 to create `dilation_square(3)(3)`; this processor dilates each input image sample by three pixels in line, and then dilates it by three pixels in column. Each pixel in the output image is thus the maximum value of the corresponding neighborhood of 9 pixels in the input image.

Listing 1.4. Dilation by a 3x3 square structuring element in Vector Faust

```
dilating(n) = strel_shift_dilation, _, strel_shift_dilation : # , _ : # : spray_by_three(n) :
      tri_maxs(n) : nconcat(n);
dilation_line(x, y) = serialize : dilating(x) : vectorize(y);
dilation_column(x, y) = matrix_transpose(y, x) : serialize : dilating(y) : vectorize(x) :
      matrix_transpose(x, y);
dilation_square(x, y) = dilation_line(x, y) : dilation_column(x, y);
```

With the operations of dilation and erosion (the dual of dilation, which shrinks shapes), an entire morphological image processing library can be constructed. As a use case, we implemented in Vector Faust the car plate identification algorithm LicensePlate, based on mathematical morphology [14, 12]; it can detect and isolate a plate in an image, as illustrated in Figure 6.

Fig. 6. LicensePlate algorithm: original image (left); detected license plate (right)

5.3 Performance

Given our goal of using Faustine as a language design test bed, no real efforts have been put into optimizing run-time efficiency. The interpreter is, in fact, unusable as is in a production setting. This is even more true when one takes into account Faust strong emphasis on very high performance, a key feature users have been counting on.

To put things in perspective and illustrate Faustine limitations, we ran both the FFT and image processing applications on an Ubuntu 12.04 LTS desktop sporting two Intel Core 2 Duo CPU E8600 64-bit processors running at 3.3 GHz each, with 3.7 GB of main memory. Dealing with a single frame of 128 64-bit floating-point numbers takes our FFT algorithm 22.4 seconds to process. A single small 195×117 image took LicensePlate 812 seconds; note that a subsequent test with a computer using a similar CPU but twice the memory size took a more reasonable 90 s to complete.

6 Future Work

The results of the previous section suggest that Vector Faust is a good candidate to express vector operations fit to perform frame-based operations, such as those present in audio spectral analysis or image processing systems. The Faustine interpreter system, as an experimental platform to run practical tests of Vector Faust programs, was instrumental in getting these results in a short period of time, proving its intrinsic value as a language design development tool. We discuss in this section some of the ideas our use of Faustine helped spur.

6.1 Vector Extension Issues

One unexpected outcome of the use of Faustine is that we found unanticipated difficulties with the current design of Faust vector extension. Since this addition to Faust is, for the time being, rather primitive, in that no higher-order constructs such as `map` or `reduce` are provided, all array operations must be specified at the element-access level, typically `a[i]`, leading to the introduction of numerous macros. For instance, Listing 1.5 implements a matrix transposition algorithm in a very straightforward manner. The block diagram resulting from the expansion of all these macros, following Faust design principle, is given in Figure 7.

Listing 1.5. Matrix transposition

```
process = matrix_transpose(3,3);
matrix_transpose(n, m) =
  _ <: par(i, n, [i]) <: par(j, m, (par(i, n, [j]) : concats(n))) : concats(m);
concats = case {
  (1) => vectorize(1);
  (n) => concats(n−1) # vectorize(1);
};
```

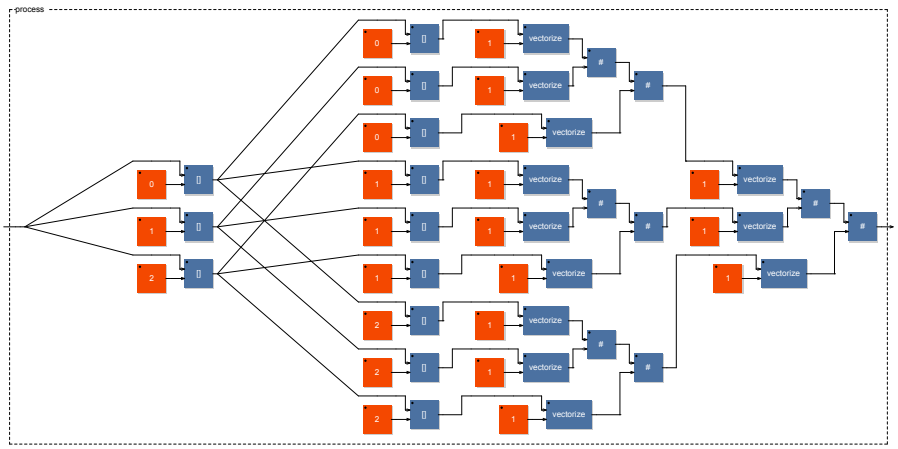

Fig. 7. Transpose diagram of a 3×3 matrix

As one can see, the `transpose` definition leads to the creation of a rather large Core Faust expression. This would have been even more patent had we used a more meaningful matrix size. In fact, when running LicensePlate, we tried to use an image of size 640×383, and the macro expansion phase of Faust original compiler (*not* Faustine per se) could not manage to complete its task, even after multiple hours. Thus addressing problems with data sets of significant size seems

to make the whole "macro/core" structure of the current Faust compilation process unusable for common array operations. Discovering this problem will affect even a future Faust-with-vectors compiler, making the introduction of higher-level constructs a necessity.

One other difficult case we encountered regards the handling of "overlapping FFT", where the successive frames for which an FFT transform is required overlap. We have not yet managed to find a totally general solution, with arbitrary overlaps, to this problem. Algorithm-specific questions such as these open opportunities for possible changes to the vector extension specification, and are at the core of what DSLs are about, i.e., finding a good match between generality and domain specificity.

6.2 Static Typechecking

One of the major limitations of Faustine, beside the lack of GUI support we already alluded to, lies in the current dynamic nature of its type checking. Signal rate and type information is currently computed and checked at run time. This may lead to run-time errors when programmers plug together unmatched signals (for instance via a "`:`" combinator). An optimizing compiler would preferably have to sport a static checker of types and rates. This is particularly true for a language such as Faust where having a precise knowledge of some of the key parameters in a program, e.g., delays, is crucial to assuring C++-like run-time performance.

Typing Vector Faust expressions is a non-trivial problem given the sophisticated nature of their type information. In particular, the presence of dependent datatypes (e.g., intervals specify the expected range of possible values of a given numeric type) is reminiscent of refinement [11] and liquid [24] types. One standard way to approach such typing systems is to use SMT solvers such as Z3[9] to handle the value-based equalities and inequalities implied by the typing rules. In addition to such tools, we envision to look carefully at the structure of constraints induced by the specifics of Faust (which does not allow first-class function values) or to design typing assistants that may ask for inputs from programmers to ensure type-checking correctness ([1, 4, 8, 19, 6]).

7 Conclusion

Faustine is a new interpreter-based test bed implemented to assess the validity of possible language extensions, in particular regarding vector operations, for the digital audio signal processing language Faust. More specifically, this platform is the first implementation of the vector/multirate extension for Faust proposed in the literature.

We used, as test cases, multidimensional FFTs and morphological image processing algorithms. These experiments suggest that the vector extension semantics can be implemented in a compliant way regarding the Faust language

[9] http://z3.codeplex.com

design. Yet, these same benchmarks show that further research is needed on the optimization front, both at the implementation and language design levels. This is paramount, given that the Faust language philosophy is to prove that a high level of expressibility is compatible with ultimate efficient run-time performance.

More generally, our design and implementation of Faustine strengthen the case for the development of interpreters, seen as flexible and easy-to-modify test beds for exploring the possible evolutionary paths of compiled languages. This idea seems to be even more convincing for highly optimized languages such as DSLs, for which introducing changes and updates within their large compiler platforms is a risky and costly proposition.

Acknowledgments

We thank Yann Orlarey for his help regarding Faust, Laurent Daverio for his input on LicencePlate and Benoit Pin for his advice on the Faustine development platform. The anonymous reviewers and Oleg Kyselyov provided many suggestions that greatly improved the quality of our paper. This work is part of the FEEVER project, partially funded by the Agence nationale de la recherche, under reference ANR-13-BS02-0008-01.

References

1. Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In *Certified Programs and Proofs*, pages 135–150. Springer, 2011.
2. Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.
3. Karim Barkati and Pierre Jouvelot. Synchronous programming in audio processing: A lookup table oscillator case study. *ACM Computing Surveys*, 46(2), 2014.
4. Thomas Bouton, Diego Caminha B De Oliveira, David Déharbe, and Pascal Fontaine. veriT: an open, trustable and efficient SMT-solver. In *Automated Deduction–CADE-22*, pages 151–156. Springer, 2009.
5. Stefan Brunthaler. Why interpreters matter (at least for high level programming languages). *http://www.ics.uci.edu/~sbruntha/why-interpreters-matter.html*, 2012.
6. Pascal Cuoq, Julien Signoles, Patrick Baudin, Richard Bonichon, Géraud Canet, Loïc Correnson, Benjamin Monate, Virgile Prevosto, and Armand Puccetti. Experience report: Ocaml for an industrial-strength static analysis framework. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 281–286, New York, NY, USA, 2009. ACM.
7. Chucky Ellison and Grigore Rosu. An executable formal semantics of c with applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 533–544, New York, NY, USA, 2012. ACM.

8. Pascal Fontaine, Jean-Yves Marion, Stephan Merz, Leonor Prensa Nieto, and Alwen Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 167–181. Springer, 2006.
9. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.
10. Debasish Ghosh. *DSLs in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010.
11. Andrew D Gordon and Cédric Fournet. Principles and applications of refinement types. *Logics and Languages for Reliability and Security*, 25:73–104, 2010.
12. Pierre Guillou. Portage et optimisation d'applications de traitement d'images sur architecture many-core. Technical report, Centre de recherche en informatique, MINES ParisTech, 2013.
13. Christoph A. Herrmann and Tobias Langhammer. Combining partial evaluation and staged interpretation in the implementation of domain-specific languages. *Science of Computer Programming*, 62(1):47 – 65, 2006. Special Issue on the First MetaOCaml Workshop 2004.
14. Jun-Wei Hsieh, Shih-Hao Yu, and Yung-Sheng Chen. Morphology-based license plate detection from complex scenes. In *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, volume 3, pages 176–179. IEEE, 2002.
15. Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es):196, December 1996.
16. Pierre Jouvelot. ML: Un langage de maquettage ? In *AFCET Workshop on New Languages for Software Engineering*, Evry, Oct. 1985.
17. Pierre Jouvelot and Yann Orlarey. Dependent vector types for data structuring in multirate Faust. *Comput. Lang. Syst. Struct.*, 37:113–131, July 2011.
18. Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml system, 1998.
19. Quang Huy Nguyen, Claude Kirchner, and Hélène Kirchner. External rewriting for skeptical proof assistants. *Journal of Automated Reasoning*, 29(3-4):309–336, 2002.
20. Yann Orlarey, Dominique Fober, and Stéphane Letz. An algebra for block diagram languages. In *Proceedings of International Computer Music Conference*, pages 542–547, 2002.
21. Yann Orlarey, Dominique Fober, and Stéphane Letz. Faust: an efficient functional approach to DSP programming. *New Computational Paradigms for Computer Music*, 2009.
22. Ariel Ortiz. Language design and implementation using ruby and the interpreter pattern. In *ACM SIGCSE Bulletin*, volume 40, pages 48–52. ACM, 2008.
23. Didier Rémy. Using, understanding, and unraveling the OCaml language from practice to theory and vice versa. In *Applied Semantics*, pages 413–536. Springer, 2002.
24. Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM.
25. Jean Serra and Pierre Soille, editors. *Mathematical morphology and its applications to image processing*. Computational Imaging and Vision. Kluwer Academic Publishers, 1994.
26. Guy L Steele and Gerald J Sussman. The art of the interpreter or, the modularity complex (parts zero, one, and two). Technical report, Cambridge, MA, USA, 1978.