Signal Processing in the Pure Programming Language

Albert Gräf
Dept. of Computer Music, Institute of Musicology
Johannes Gutenberg University
55099 Mainz, Germany
Dr.Graef@t-online.de

Abstract

This paper introduces the author's new functional programming language Pure and discusses its use as a scripting language in signal processing applications. Pure has a JIT compiler based on the LLVM compiler framework which makes execution reasonably fast and interfacing to C very easy. A built-in GSL matrix type makes it possible to handle numeric signals in an efficient way, and a Pure plugin for Miller Puckette's Pd provides the necessary infrastructure for realtime signal processing. The paper gives a brief overview of the language and the Pd interface, and presents some examples.

Keywords

Functional programming, Pd, signal processing, term rewriting.

1 Introduction

Programming signal processing in imperative programming languages is difficult and error-prone. Functional programming (FP) has long been identified as a viable alternative. Signals are just functions of time after all, and FP allows us to formulate the "patching" of signal processing components very conveniently through higher-order functions. Another important benefit of this approach is that purely functional programs have simpler semantics and can thus serve as formal and platform-independent specifications of signal processors, which is an important prerequisite for building a portable library of standard signal processing components.

So, if FP offers such striking advantages, then why don't we all use it? One major obstacle is that, to make those specifications executable, typically quite a bit of glue code is needed, and most FP languages make it annoyingly difficult to interface to the "imperative world" out there. Static typing, which is enforced by most modern FP languages, raises yet another barrier, since ad hoc message protocols cannot be represented conveniently in such frameworks, even if they support parametric polymorphism.

The Pure language attempts to overcome these problems. Pure is loosely based on the author's Q language [1], but it is a much improved design with a completely new implementation offering JIT (just in time) compilation to native code using the LLVM compiler framework (`http://llvm.org`). Like Q, Pure is based on term rewriting which makes it a powerful tool for developing algebraic models. Such models play an important role in recent approaches to formalize signal processing systems [2; 4]. In addition, Pure offers the following features:

- Since Pure is *dynamically typed*, interfacing to interpreted realtime environments such as Pd and SuperCollider is much more convenient than in statically typed languages.
- Pure has a built-in Octave-like *matrix type*, which can be used to represent numeric signal data in an efficient way and is compatible with the GNU Scientific Library.
- As a JIT-compiled language, Pure is suitable for applications where many interpreted languages fall short, but it also offers a convenient interactive environment just like a "real" interpreter.[1]
- You can call any C function *directly* in Pure, so the glue code necessary to interface to the environment can often be written in Pure itself.
- A plugin for Miller Puckette's Pd gives access to a graphical environment for testing realtime signal processing applications.

[1] Taking the Alioth "recursive" benchmark as an example (`http://shootout.alioth.debian.org`), Pure comes out somewhere between MZScheme 4.1.3 (which is about 6 times faster) and Python 2.5 (which is about 6 times slower). This still leaves much room for improvements, but note that MZScheme is a *very* fast and mature interpreter. To put these into perspective, Q is about 26 times slower than Pure on this benchmark.

The paper gives a brief overview of Pure and its Pd interface, and we also discuss two simple examples for actor-style audio and stream-based message processing. Finally, we point out some open problems and directions for further research.

Note: The Pure interpreter is free software distributed under the GPL V3. Please visit the Pure website at: `http://pure-lang.googlecode.com`

2 Pure in a Nutshell

Like its predecessor Q, Pure is based on term rewriting, so its data objects are *terms* (expressions) which are rewritten according to the symbolic equations supplied by the programmer. Terms can be atomic objects like numbers, strings and symbols, from which compound terms are formed using *function application*. Application is written simply as juxtaposition: $f\,x$, where f is the function and x the argument. Application associates to the left, i.e., $f\,x\,y = (f\,x)\,y$, and thus a function of two arguments is actually a function of the first argument which yields another function operating on the second argument. This style of writing function applications is also called *currying*, and can be found in many modern FP languages.

Definitions take the form of equations which are always applied from left to right. For instance, here is a definition of the well-known factorial function:

```
fact n = if n>0 then n*fact(n-1) else 1;
```

Pure also provides guarded equations, so the above example can also be written as follows:

```
fact n = n*fact(n-1) if n>0;
       = 1 otherwise;
```

You can either put such definitions into a script which is then loaded in the interpreter, or just start up the interpreter and type the definitions at its command prompt '`>`':

```
> fact n = if n>0 then n*fact(n-1) else 1;
> map fact (1..10);
[1,2,6,24,120,720,5040,40320,362880,3628800]
```

The second input line above applies our function to the list of integers between 1 and 10, which returns the list of the first ten factorial values. Expressions are generally evaluated from left to right, innermost expressions first, until a *normal form* is reached which cannot be simplified any further by applying equations; the resulting term is then the value of the original expression, which is what gets printed by the interpreter.

New operators can be defined just as easily. For instance, the following code defines a left-associative infix operator '`over`' which computes the binomial coefficient $\binom{n}{k}$:

```
> infixl 5 over;
> n over k = fact n div (fact k*fact (n-k));
> 10 over 5;
252
```

The predefined operators like '`*`' and '`div`' are in fact not special; they are all defined in Pure's *prelude*, which is just an ordinary Pure script that gets loaded by the interpreter at startup. (The compiler does provides some special support for these and other arithmetic and logical operations so that they can be translated to efficient native code when they are applied to the appropriate arguments. But this is just an optimization in the generated code and does not affect the semantics of these operations.)

Operators are really just ordinary functions in disguise. By enclosing an operator in parentheses, you can turn it into an ordinary prefix function; e.g., `n over k` is the same as `(over) n k`. Moreover, functions are first-class and can thus be passed around just like any other values. For instance, here is how we can compute the 10th row of the Pascal triangle:

```
> map ((over) 10) (0..10);
[1,10,45,120,210,252,210,120,45,10,1]
```

Note the *partial application* of '`over`' to `10`, which yields a function mapping `over`'s second operand `k` to `10 over k`. An alternative way to derive such little anonymous functions is the *lambda abstraction*:

```
> map (\k -> 10 over k) (0..10);
[1,10,45,120,210,252,210,120,45,10,1]
```

Using so-called *list comprehensions*, this can also be written as follows:

```
> [10 over k | k = 0..10];
[1,10,45,120,210,252,210,120,45,10,1]
```

Matrices work pretty much like lists, but are written in curly braces instead of brackets. Different rows are separated with semicolons:

```
> {1+2,2+3,4+5};   // a row vector
{3,5,9}
> {1,2+3,4;5,6,7}; // a 2x3 matrix
{1,5,4;5,6,7}
```

As indicated, vectors are represented as single-row matrices. Matrix comprehensions are provided, too. For instance, here is how we can employ the `rand` function from the C library to generate some white noise:

```
> extern int rand();
> randf = rand/0x7fffffff;
> noise n = {2*randf-1 | i = 1..n};
```

The prelude provides an extensive collection of generic list and matrix operations, such as indexing, slicing, mapping, accumulating, etc. Additional matrix operations are supplied by the `gsl` module. Integer and floating point matrices are internally represented in a format which allows data pointers to be passed directly between Pure and C. This makes it possible to handle copious amounts of numeric data in an efficient way, which is an important requisite for signal processing applications.

So far we have only dealt with numbers and simple aggregates of these. But Pure also makes it easy to handle symbolic data. To these ends, function symbols without any defining equations can be used as data *constructors*. For instance, lists are in fact represented using the binary infix symbol '`:`', which does not have any defining equations and thus acts as a constructor symbol. The term `x:xs` denotes a list with head element `x` and tail `xs`. Thus, `[1,2,3]` is just syntactic sugar for `1:2:3:[]`, where `[]` is the empty list. To dissect such structures, function definitions may involve *pattern matching* on the arguments. E.g., a function to add up all values in a list can be defined as follows:

```
> sum (x:xs) = x+sum xs; sum [] = 0;
> sum (1..10);
55
```

Custom data structures can be dealt with just as easily. For instance, suppose that we represent note events as terms of the form `note n v`, where `note` is the constructor symbol signifying a note event, and `n` and `v` are integers denoting the note number and velocity, respectively. Then we can define an operation to transpose a note event (i.e., shift the note number by a given number of semitones) simply as follows:

```
> trans k (note n v) = note (n+k) v;
> trans 5 (note 60 64);
note 65 64
```

Note that this definition only applies to terms of the form `note n v`. Pure has no problem dealing with such partial definitions; if there is no equation for a given combination of arguments, the subject term is in normal form, i.e., it evaluates to itself. For instance, with the above definition of `trans` you'll get:

```
> trans 5 60;
trans 5 60
```

New equations can be added at any time, to deal with cases that have not been covered before. For instance, we can make `trans` work on stand-alone note numbers as follows (the '`::int`' tag in this equation is a special kind of pattern which restricts the argument `n` to be an integer):

```
> trans k n::int = n+k;
> trans 5 60;
65
```

What this means is that functions are always *polymorphic* in Pure, i.e., they may apply to as many different types of arguments as you like. Like Lisp and Prolog, Pure is essentially a *typeless* language; all data belongs to the same universe of terms, and the equations alone determine which functions can be applied to which arguments.

We mention in passing that Pure's symbolic rewriting capabilities actually go beyond what can be found in most FP languages, since Pure does not distinguish between "defined" and "constructor" function symbols, and thus any kind of function (or operator) symbol is permitted inside patterns. For instance, suppose that we want lists to automatically stay sorted and eliminate duplicates. In Pure we can do this by simply adding the following equations:

```
> x:y:xs = y:x:xs if x>y;
>        = x:xs if x==y;
> [13,7,9,7,1]+[1,9,7,5];
[1,5,7,9,13]
```

Thus Pure lets you deal with algebraic simplification rules and constructor equations in a direct fashion. Therefore we consider Pure not just a functional, but also an *algebraic* programming language, akin to languages such as Maude, OBJ and OPAL, although it is geared more to practical applications than these.

For the sake of expressivity, Pure actually extends the basic term rewriting machinery sketched out above in many ways. In particular, expressions and equations can be annotated with nested local function (`with`) and variable definitions (`when`). For instance, here is an implementation of the Fibonacci function which employs a local helper function to compute the Fibonacci numbers in pairs:

```
fib n = a when a,b = fibs n end with
  fibs n = 0,1 if n<=0;
         = b,a+b when a,b = fibs (n-1) end;
end;
```

Pure also has exception handling, macros, modules and namespaces, and thus offers all the necessary means to structure programs both in the small and in the large. We cannot cover these facilities in this paper, so please refer to the Pure manual for details.

3 Pure and Pd

We will use Miller Puckette's Pd as the basic signal processing machinery, so that we do not have to worry about the technical intricacies of realtime processing and multimedia protocols. A `pd-pure` plugin is available on the Pure website, which lets us implement Pd control objects in Pure. To enable this plugin in your Pd patches, you need to tell Pd to load the plugin at startup, which can be done either with Pd's `-lib` option, or by adding `pure` to Pd's startup options. (The latter setting can be saved so that the plugin is always loaded when you run Pd.)

`pd-pure` is actually a Pd *loader* which looks for the definitions of Pd control objects in corresponding Pure scripts. For instance, if you create an object named `foo` in Pd, it will try to find a Pure script `foo.pure` in the directory of the patch; if found, it will load the script and evaluate the `foo` function (presumably defined in `foo.pure`) to obtain the *object function* which implements the Pd control object. In the simplest case, the object function can just be `foo` itself, and `pd-pure` equips the object with a single inlet and a single outlet. Pd messages are passed from the inlet as parameters to the object function, and function results are sent to the outlet. (It is also possible to create objects with any number of inlets and outlets, see the `pd-pure` documentation for details.)

Pd messages are translated to corresponding Pure expressions and vice versa in a straightforward fashion. Special support is provided for converting between the natural Pd and Pure representations of floating point numbers, symbols and lists. The following table summarizes the available conversions.

Message	Pd	Pure
symbol	`bar`	`bar`
float	`float 1.23`	`1.23`
list	`list 1 2 3`	`[1.0,2.0,3.0]`
other	`bar a 2 3`	`bar a 2.0 3.0`

For instance, the following object accepts Pd `float` messages and adds the first parameter `x` (which is assumed to be supplied at object creation time) to each received value:

```
add x y = x+y;
```

After placing this code in a file `add.pure`, you can invoke the function from Pd as shown in Fig. 1.

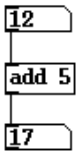

Figure 1: A simple Pure object.

4 Actors versus Stream Processing

Pd objects implement a dataflow paradigm known as *actor*-style processing. Each actor realizes a simple signal processing task, and complex systems are built by connecting the individual actors so that data flows from one actor to another. This approach is very intuitive, especially when implemented in a graphical system like Pd. However, specifying complex iterative and recursive signal processes can be quite difficult in this model.

Also, actors typically encapsulate local state to accomplish their task (e.g., think of counters which have to keep track of the current value). Such actors are no functions in the mathematical sense any more, because the output of an actor not only depends on its inputs but also on its internal state. This makes formal reasoning about such systems difficult (ever tried to figure out what a complex Pd or Max patch actually *does*?).

While Pure is capable of implementing stateful actors, there is an alternative approach which fits the functional programming model much better. The basic idea is to view the entire sequence of actions performed by an actor as a function which maps a stream of input messages to another stream of output messages. This *stream processing* approach can be traced back to early functional dataflow languages such as Lucid (Wadge, 1976). A related method, which is due to the Yale Haskell group, has become known under the catchy term *functional reactive programming* (`http://www.haskell.org/frp`).

In FP terminology, a *stream* is a "lazy" list whose tail is left unevaluated until it is actually needed. This is essential for our purposes, because the elements of the input stream are not known in advance; they only become available in realtime during execution. In Pure such

lazy data structures are constructed with the '`&`' postfix operator, which changes the order of evaluation. It does this by turning its operand into a *thunk* (also called a *future* in FP parlance), a kind of parameterless function whose evaluation is deferred until the corresponding value is needed. For instance, a Pd counter object can be implemented in a completely stateless way as follows:

```
counter = process (1..inf) with
  process (n:ns) (bang:next) =
    n : process ns next &;
end;
```

Note that `1..inf` is the infinite stream of all positive integers. In this example, the `process` function responds to each `bang` message by producing the next counter value. The rest of the computation (the "future") is then handled by the thunked recursive invocation of `process` in the tail of the output stream.

Finally, we need to consider how to interface a stream processing function like the one above to Pd's actor-based model. The problem is that the Pd object is expected to take single Pd messages as input and produce single messages as output, not streams of such messages. Thus we need a kind of "adapter" function which keeps track of incoming messages, feeds them as a stream to the stream processing function, polls the output stream for results and sends the produced messages through the object's outlet.

This is a bit tricky to do right now, because Pure does not yet support multithreaded programming. A possible solution, which uses exceptions to implement the polling of the input stream in a transparent, non-blocking way, is given by the `actor` function in Fig. 2. This function takes a stream processing function and returns an equivalent actor function ready to be used in Pd and similar environments. For brevity, we just list this here without further explanation; a commented version of this routine can be found in the accompanying materials.[2]

5 Audio Processing Example

Let us first have a look at a simple actor-style example illustrating how we can read chunks of audio data from a soundfile and pass that data to Pd so that it can be played back in realtime. This example also demonstrates how easily we can interface to existing C libraries in Pure, without writing a single line of C code.

Reading the soundfile is accomplished using Erik de Castro Lopo's well-known `libsndfile` library (see Fig. 3). The `using` clause at line 1 instructs the interpreter to "dlopen" `libsndfile.so`, the remaining `extern` lines are C prototypes telling the interpreter about the C routines that we need. After these declarations the Pure interpreter knows how to call these functions.

The `sf_load` routine at line 7 is a little wrapper function around `sf_open` which provides some temporary storage for the `info` parameter required by `sf_open`. It also installs a finalizer ("sentry" in Pure parlance) on the resulting file pointer, so that it gets closed automatically when the file object is no longer needed.

The `wavefile` object itself is defined at line 14. It responds to the message `reset` by rewinding the soundfile using the `sf_seek` function, and to the message `bang` by producing the next chunk of audio data. `pd-pure` does not support audio objects written in Pure right now, but it does provide some special C routines to transfer audio data between Pure vectors and Pd arrays. Here we read the audio data with the `sf_read_double` function into a Pure vector and then transfer the data to a Pd array using `pd_setbuffer`. Finally, we check the result of `sf_read_double`, and output a `bang` message if we got some new data, to indicate that the data is ready to be processed by the hosting Pd patch. Note that the other branches just return an empty tuple `()` indicating a "non-value" which does not cause any output.

The patch shown in Fig. 4 demonstrates the use of the `wavefile` object in Pd. The `pd play` subpatch (not shown in the figure) contains the necessary logic to supply the `wavefile` object with a steady stream of `bang` messages if the toggle control is activated. Switching the toggle off causes a `reset` message to be emitted, which resets the file pointer to the beginning of the file. The patch uses Pd's `tabplay~` object to play back the data, which takes `bang`'s from the `wavefile` object to trigger the playback of each chunk of audio data as it becomes available.

6 Reactive Animation Example

Our second example is a little "bouncing ball" animation which uses Pd's OpenGL interface Gem to realize the graphics. The animation has some interactive elements to change the posi-

[2] A tarball with the examples discussed in this paper can be found at `http://pure-lang.googlecode.com/svn/docs/pure-lac09.html`.

```
nullary nodata;
actor f = process with
  process ()          = digest [];
  process msg         = digest [] when put q (get q+[msg]) end;
  digest msgs         = catch (check msgs) (retrieve msgs);
  retrieve msgs       = case get r of
    y:ys = digest (y:msgs) when put s (); put r ys end;
    ys = if null msgs then throw (bad_list_value ys) else return msgs;
  end;
  check msgs nodata   = return msgs;
  check _ x           = throw x otherwise;
  return msgs         = [x | x = reverse msgs; x~==()];
end
when s = ref (inputs&); r = ref (f (get s)&) end
with inputs = case get q of
                x:xs = x:inputs& when put q xs end;
                _ = throw nodata;
              end;
end when q = ref [] end;
```

Figure 2: Interfacing between actor-style and stream-based processing.

```
 1  using "lib:libsndfile";
 2  extern SNDFILE* sf_open(char *path, int mode, SF_INFO *sfinfo);
 3  extern int sf_close(SNDFILE *sndfile);
 4  extern long sf_seek(SNDFILE *sndfile, long frames, int whence);
 5  extern long sf_read_double(SNDFILE *sndfile, void *ptr, long items);
 6
 7  sf_load name::string = sentry sf_close (sf_open name 0x10 (imatrix 10));
 8
 9  extern int pd_getbuffersize(char *name);
10  extern void pd_setbuffer(char *name, expr* x);
11
12  nullary bang reset;
13
14  wavefile fname aname = process with
15    // Rewind to the beginning of the file.
16    process reset = () when sf_seek fp 0 0; end;
17    // Read the next chunk of audio. Output bang if ok.
18    process bang  = if ok res then bang else () when
19      n = nsamples; wave = dmatrix n;
20      res = sf_read_double fp wave n;
21      pd_setbuffer aname wave;
22    end;
23    nsamples      = pd_getbuffersize aname;
24    ok res        = bigintp res && res>0;
25  end when
26    fp::pointer = sf_load fname;
27  end;
```

Figure 3: Reading soundfiles.

tion, velocity and acceleration of the ball while the animation is running. In contrast to the previous example, we are going to implement this one in a purely functional way using the stream processing approach sketched out in Section 4.

Using classical mechanics, the motion of an accelerated object in two dimensions can be defined as a stream processing function as follows:

```
motion (x,y) (vx,vy) (ax,ay) (step dt:next)
= [x,y,vx,vy] :
  motion (x1,y1) (vx1,vy1) (ax,ay) next &
```

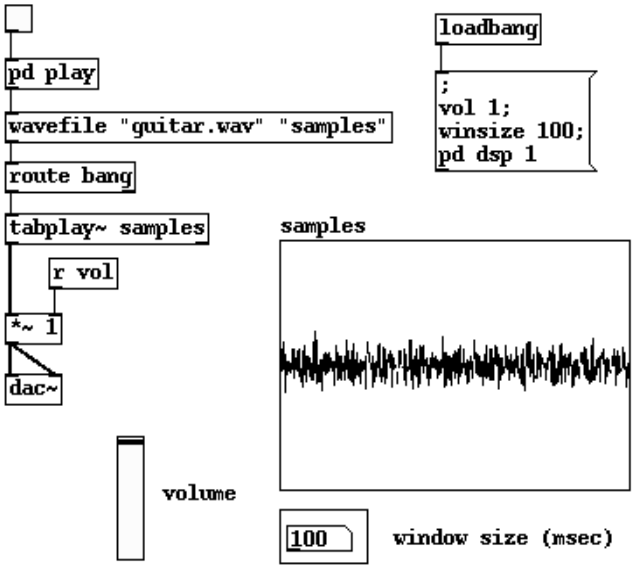

Figure 4: The `waveplay` patch.

```
when
  vx = if abs x > 3 then -vx else vx;
  vy = if y < -3 then -vy else vy;
  x1 = x+dt*vx+dt*dt*ax/2;
  y1 = y+dt*vy+dt*dt*ay/2;
  vx1 = vx+dt*ax; vy1 = vy+dt*ay;
end;
```

The pairs `(x,y)`, `(vx,vy)` and `(ax,ay)` give the current position, velocity and acceleration in the x and y directions, respectively, and the last argument of `motion` is the stream of incoming messages. We assume that we will receive a message of the form `step dt` to produce the current x and y coordinates and velocities of the ball. The `dt` value indicates the time between `step` messages, which is used to update position and velocity for the next `step` message. In addition, the first two definitions in the `when` clause take care of the collision detection, which ensures that the ball gets reflected by "walls" at $x = -3$ and $x = 3$ and the "floor" at $y = -3$. As required, the result of the function is a stream, which has the response (a list with the current position and velocities) in front, followed by the rest of the stream, a thunked recursive invocation of `motion` with the updated parameters.

It is now an easy matter to add more equations which deal with messages to change the position, velocity and acceleration of the moving object. As an example, the following equation shows how to update the position of the ball and reset the vertical velocity in response to a mouse event.

```
motion _ (vx,vy) (ax,ay) (mouse x y:next)
= () : motion (x,y) (vx,0) (ax,ay) next &;
```

We have omitted the remaining (trivial) equations for brevity, but you can find the full source code in the accompanying materials.

To see our stream processing function in action, we can simply feed it an endless stream of `step` messages and grab some of the results:

```
> in = repeat (step 0.1);
> motion (-3,3) (0.5,0) (0,-3) in!!(0..2);
[[-3,3,0.5,0],[-2.95,2.985,0.5,-0.3],
 [-2.9,2.94,0.5,-0.6]]
```

Using the `actor` function from Section 4, the function `ball` realizing the bouncing ball object can now be implemented as follows:

```
ball = actor
       (motion (-3,-3) (0.5,0) (0,-3));
```

Fig. 5 shows a Pd patch for rendering the animation. You can also find this patch in the accompanying materials for your perusal, along with a second Pd patch which adds some sound effects to the animation.

7 Conclusion

Pure's algebraic programming style probably appeals most to mathematically inclined programmers. Nevertheless, we tried to strike a balance between mathematical purity and practicality which will hopefully make Pure useful also for non-mathematicians. In particular, Pure's dynamic typing, the seamless integration with C and the interpreter-like environment

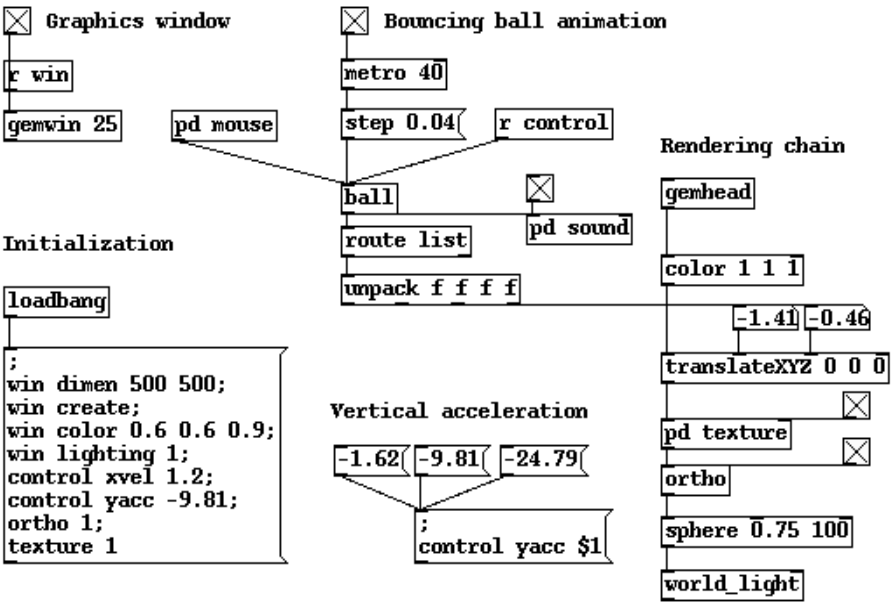

Figure 5: Animation patch.

makes it usable as a "compiled scripting language" for many purposes. This requires some compromises (Pure isn't really all *that* pure), but the author subscribes to the philosophy that a bird in the hand is worth two in the bushes.

In the signal processing realm, Pure seems especially suited for asynchronous, event-based processing of the kind found in graphical user interfaces, animations, games, computer music and robotics. But we also plan to improve the support for plain audio processing, by providing an interface to Yann Orlarey's Faust language [4], and extending the existing Pd interface so that audio processing objects can be implemented in a direct fashion.

It goes without saying that realtime signal processing is one area which can benefit tremendously from parallel processing, especially as multiprocessor systems are finally becoming ubiquitous. This is one feature which Pure still lacks right now. We are currently investigating different approaches ranging from simple data parallelism to full multiprocessing models. Pure already has lazy futures à la Alice ML, so *concurrent futures* would be one suitable model [3]. Parallelization of array comprehensions is another promising avenue.

Another important direction for future research is the development of a high-level functional framework for specifying asynchronous signal processes. This is an area of active research, and the jury is still out on what approach works best. Reactive systems are used in a wide variety of different applications, each with their own peculiarities. We think that Pure will prove to be a useful tool to study these applications and experiment with different algebraic models, which might lead to interesting new solutions.

Acknowledgements

Thanks are due to Claude Heiland-Allen for his explanations of the Pd loader interface, and to Yann Orlarey and the anonymous referees for helpful comments.

References

[1] A. Gräf. Q: A functional programming language for multimedia applications. In *Proceedings of the 3rd International Linux Audio Conference*, pages 21–28, Karlsruhe, 2005. ZKM.

[2] P. Hudak. An algebraic theory of polymorphic temporal media. Technical Report RR-1259, Yale University, Dept. of Computer Science, 2003.

[3] J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science*, 364(3):338–356, Nov. 2006.

[4] Y. Orlarey, D. Fober, and S. Letz. Syntactical and semantical aspects of Faust. *Soft Computing*, 8(9):623–632, 2004.