

hsxt

Haskell in Pd

Claude Heiland-Allen
GOTO10
claudiusmaximus@goto10.org

ABSTRACT

`hsxt` is a Pd external library for loading and using within Pd code written in the Haskell¹ programming language.

Haskell is a non-strict, purely-functional programming language, with polymorphic typing, lazy evaluation and higher-order functions.

This paper aims to introduce and discuss the motivation, implementation and use of `hsxt`, and also present a roadmap for future development.

1. INTRODUCTION

The birth of `hsxt` can be traced to late 2006, when the author returned to Haskell after several years' absence. Development of `hsxt` started in January 2007, and `hsxt-0.1` was tagged on 14th March 2007.

`hsxt-0.1` is sufficient to write simple externals, such as a generic `Swap` object that can swap arbitrary messages (Pd's internal `swap` is specific to `float` messages).

2. MOTIVATION: WHY HASKELL?

C, Python, Java, and so on, are all imperative languages, wherein a program consists of a sequence of commands describing how to compute something. Haskell is a pure functional language, wherein a program consists of a single expression describing what needs to be computed. This focus on the high-level "what" rather than the low-level "how" is a distinguishing characteristic of functional programming languages.

Functional programming languages have several features that repeal the limitations of imperative languages. For example, higher-order functions to manipulate functions, and partial application to generate more specialized functions from existing functions. These can be exemplified by:

```
map :: (a -> b) -> [a] -> [b]
(*) :: (Num a) => a -> a -> a
map (42*) [1..4] -- evaluates to [42,84,126,168]
```

¹<http://www.haskell.org>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Pd Convention 2007. Montréal, Québec, Canada.

Copyright 2007. Copyright remains with the author(s).

in which `map` applies a function to every element of a list, and `(42*)` is a partial application of `(*)`, turning a binary operator into a unary function.

Haskell is a "pure" functional language, having an explicit separation of "execution of actions" (which may have side effects) and "evaluation of expressions" (which may not). This purity (which is not present in some functional languages such as Lisp and Scheme) allows a good compiler to make seemingly astonishing optimizations² because code can be reordered and fused without changing the semantics.

Haskell is non-strict, using lazy evaluation by default. This allows infinite data structures to be defined, and evaluated just as much as is necessary (but no more). The classic example is the simple definition of the Fibonacci numbers³ by:

```
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

A benefit of lazy evaluation is the ease of implementing a producer/consumer code model: one can define a producer without worrying in advance how much needs to be produced, this task is delegated to the consumer. For example, an iterative numerical method can be modularised to a producer which generates successive approximations, and a consumer which picks the first approximation that is within the desired accuracy [3].

3. IMPLEMENTATION: WEAVING FROM C TO HASKELL AND BACK AGAIN

Haskell has a Run Time System (RTS) which must be started before any Haskell code can be used [4]. When compiling standalone programs this is handled automatically by the Haskell compiler, but for shared libraries such as Pd externals there is not such compiler support (as of `ghc-6.6`, the current stable release of the most prevalent Haskell compiler). This prevents `hsxt` being written solely in Haskell, and so there is a small amount of C code that starts the RTS and calls the main `hsxt` entry point (written in Haskell) (Figure 1).

Haskell has a Foreign Function Interface (FFI) which allows Haskell to call C functions, and also allows Haskell functions to be "wrapped" into C function pointers that are able to be called from C code [5]. The tool `hsc2hs` helps write Haskell to manipulate C values (including defines and

²"Compiling with `-O2` reduced the time taken by my program's execution from 28 mins to 17 seconds." [1]

³0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... where $f_n = f_{n-1} + f_{n-2}$ [2]

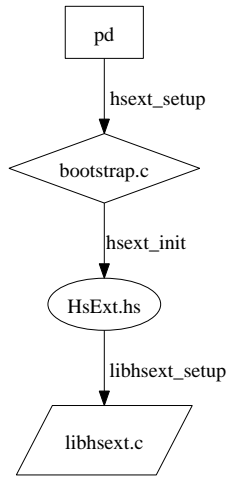


Figure 1: Control flow when loading hsex.

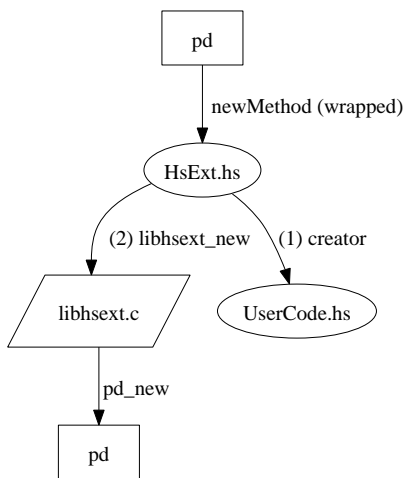


Figure 2: Control flow when creating a hsex object.

structs), but the author found it more comfortable to write small helper functions (`libhsex`) in C to abstract some of the more idiosyncratic aspects of Pd’s API from the Haskell part of `hsex`.

Haskell has a package for dynamically compiling and loading Haskell code at runtime (*Plugins*). When an `hsex` object is instantiated in Pd, the first creation argument is used to form the name of a Haskell source code file to compile and load. This Haskell source code defines a “resource”, which must be of type *Creator* (which is defined by the *PureData* API provided by `hsex`), and the remaining creation arguments are passed to it. The creator resource in `hsex-0.1` can reject the arguments, or it may return an *Instance* specifying the number of inlets and outlets, and a single function, which will handle all messages arriving at the inlets (Figure 2).

Pd’s API has several functions to create inlets and add methods, but in `hsex-0.1` an object has but one method (a design decision that later turned out to be a mistake). If the *Creator* returns an *Instance* successfully, `hsex` calls out to `libhsex` to create a new Pd object with the required number of proxy inlets (a technique used in Pd’s `list` family

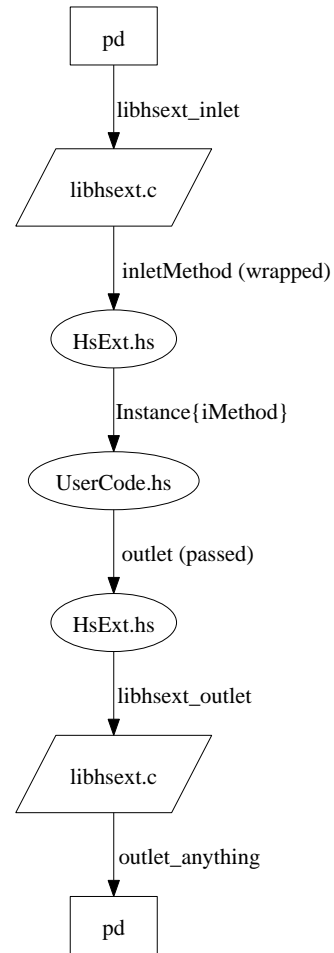


Figure 3: Control flow when data arrives at an hsex inlet.

of objects) and the required number of outlets, then links the Pd object with the *Instance* and returns the Pd object to the caller (the main Pd program).

When the Pd object receives a message at one of its proxy inlets, Pd calls a function in `libhsex`. This function looks into the inlet structure to find the owning *Instance* ID, and passes all this information into the Haskell core of `hsex`. The core calls the one method defined in the *Instance* with these arguments, and in addition passes in a structure exposing the *PureData* API. In `hsex-0.1` this last is a function that can be used to send messages to the Pd object’s outlets (and thus back into the C environment of the Pd main program) (Figure 3).

4. USING HSEX: A GENERIC SWAP OBJECT

The simplest not-quite-trivial example showing most of the features of `hsex-0.1` follows. The task is to build a Pd object that behaves like Pd’s internal `swap`, but that can operate on arbitrary messages, not just `float` messages. It can be named `Swap`, stored in a file called `Swap.hs`, and instantiated in Pd like `[hsex Swap]`.

First, we need a module declaration:

```
module Swap (creator) where
```

Next, we need to import the *PureData* API:

```
import PureData
```

We need to preserve state between calls from Pd, and a simple way to do this is using *IORefs*:

```
import Data.IORef
```

Now we can define our creator resource, which creates a new initial state and returns an *Instance*. Notice how the state variable reference is embedded in the returned method:

```
creator :: Creator
creator _ = do
  bang ← gensym "bang"
  state ← newIORef (bang, [])
  return (Just Instance {
    iInlets = 2,
    iOutlets = 2,
    iMethod = inlet state })
```

The left inlet recalls the stored state (a message), sends the incoming message to the right outlet, and sends the recalled message to the left outlet:

```
inlet state outlet 0 s m = do
  (rs, rm) ← readIORef state
  outlet 1 s m
  outlet 0 rs rm
```

The right inlet simply stores the incoming message:

```
inlet state outlet 1 s m = do
  writeIORef state (s, m)
```

5. THE NEXT LEVEL: TOWARDS HSEXT-1.0

The *PureData* interface defined by *hsext-0.1* is deficient, for several reasons. Firstly, the “creator resource” implies that a Pd object written in Haskell cannot have a state shared between its instances⁴. Moreover, the “creator resource” implies that each Haskell source file can define only one Pd object class. In *hsext-1.0* the “creator resource” will be replaced with a “setup resource”, which *hsext* will guarantee is executed exactly once (thus allowing state unique to each source file).

Using the `[hsext RealNameOfClass]` idiom to load Haskell code is ugly. Using Pd’s `sys_register_loader`, will give “full” class status to Haskell objects, making `[Swap]` possible where before `[hsext Swap]` was required [6].

Next, the “one method for all inlets” paradigm is mismatched to Pd’s way of doing things, making it awkward to have different behaviour for different selectors. *hsext-1.0* will map to Pd’s API more directly from *hsext*’s *PureData* interface, and it will map to a larger proportion of Pd’s API too (for example, clock callbacks, receiver names, and table access).

Finally, the way the “outlet function” is passed to the object’s inlet method is non-extensible – extra functions may be desired in future *hsext* versions – but some passing in

⁴Haskell provides functions such as *unsafePerformIO* which might be used to work around this, but these are fraught with danger.

of functions is required, given the absence of safe global mutable state in Haskell. A more extensible solution is using Haskell records: passing in a record of functions to the setup resource. The Haskell core of *hsext* would embed references to its mutable state inside the functions in the record, there remains the notational inconvenience at the point of use, namely having to write `(apiCall pd) x y z` instead of `apiCall x y z`.

The proposed API can be found in the appendix (Figure 5).

6. USING HSEXT-1.0: GENERIC SWAP REVISITED

The *Swap* written for *hsext-0.1* needs to be rewritten for *hsext-1.0*:

```
module Swap (setup) where
import PureData
import Data.IORef
import qualified Data.Map as M
```

The “setup resource” needs to store the Pd class pointer: Pd doesn’t pass it to the constructor, yet the constructor needs it to create the Pd object. Moreover, there is a need to associate Pd object pointers to Haskell object states, because the destructor needs to free data stored in Haskell-space but the Pd defines destructors on a per-class rather than a per-object basis. The *IORef* stores a *Maybe t* value to emulate emptiness – the *IORef* must be created before use, and the only sensible value to fill it comes after its use in the partial application of the constructor and destructor.

```
setup pd = do
  nameS ← (gensym pd) "Swap"
  classR ← newIORef Nothing
  classP ← (newClass pd) (nameS)
             (new classR) (free classR)
  writeIORef classR (Just (pd, classP, M.empty))
```

Haskell’s strong static typing prevents the type-casting tricks used in Pd’s C API – to emulate the behaviour of Pd’s `class_new` relating to creation argument types would require a function for every possible list of argument types – and so *hsext* uses only Pd’s `A_GIMME`, losing a little convenience for *hsext* users but avoiding the combinatorial explosion.

The constructor creates the Pd object together with its inlets (and methods) and outlets. This implementation ignores the creation arguments and the initial stored value is an empty `bang` message.

```
new classR _ = do
  Just (pd, classP, m) ← readIORef classR
  objectR ← newIORef Nothing
  stateR ← newIORef (bang pd, [])
  objectP ← (newObject pd) classP
  inlet0 ← (newInlet pd) objectP ≫=
            (addAnything pd) (swap0 objectR)
  inlet1 ← (newInlet pd) objectP ≫=
            (addAnything pd) (swap1 objectR)
  outlet0 ← (newOutlet pd) objectP
  outlet1 ← (newOutlet pd) objectP
  writeIORef objectR
    (Just (pd, stateR, inlet0, inlet1, outlet0, outlet1))
  writeIORef classR
```

```
(Just (pd, classP, M.insert objectP objectR m))
return objectP
```

The destructor uses the Pd object pointer to find the associated Haskell object state within the Haskell class state. The Haskell class state must be updated with this association removed to avoid a memory leak.

```
free classR objectP = do
  Just (pd, classP, m) ← readIORef classR
  objectR ← M.lookup objectP m
  writeIORef classR
    (Just (pd, classP, M.delete objectP m))
  Just (←, ←, inlet0, inlet1, outlet0, outlet1)
    ← readIORef objectR
  (freeOutlet pd) outlet1
  (freeOutlet pd) outlet0
  (freeInlet pd) inlet1
  (freeInlet pd) inlet0
  (freeObject pd) objectP
```

The first inlet is “hot”, reading the state and triggering outlet actions.

```
swap0 objectR msg = do
  Just (pd, stateR, ←, ←, outlet0, outlet1)
    ← readIORef objectR
  state ← readIORef stateR
  (outAnything pd) outlet1 msg
  (outAnything pd) outlet0 state
```

The second inlet is “cold”, storing the incoming message in the state.

```
swap1 objectR msg = do
  Just (←, stateR, ←, ←, ←, ←)
    ← readIORef objectR
  writeIORef stateR msg
```

The proposed new API is more powerful, but this flexibility comes at a price. The generic `Swap` rewritten to the proposed `PureData` API is significantly more verbose. However, if there is a need for many objects meeting the same type-signature (for example, the family of binary numeric operators `[+]` `[-]` `[*]` `[/]` and so on), it is possible to write a wrapper for that type-signature and define each required class of that signature by a single statement.

7. FUTURE WORK: SENDING HASKELL DATA THROUGH PD PATCH-CORDS

Haskell data could be sent through Pd patch-cords by several means. One could convert a Haskell value to a `StablePtr` (which prevents the garbage collector from moving the value to a different location in memory) and (mis)use Pd’s pointer atom type. However, this incurs a space leak, as `StablePtrs` must be deallocated manually and there is no way of knowing which non-Haskell objects are still storing them; moreover the Pd objects that expect pointer atoms expect Pd data structures to be pointed to by them, and receiving something other than that would likely lead to a crash. Thus this approach is ruled out.

Another approach is much more robust but has some caveats. One could store the value in Haskell-space, and send something different through the Pd patch-cords, which when arriving at another hsex object would tell that object

```
module Dispatcher (
  Dispatcher, -- abstract data type
  empty,     -- an empty dispatcher
  fallback,  -- for unmatched types
  insert,    -- add type method
  delete,    -- delete type method
  dispatch   -- dispatch type method
) where
import Data.Dynamic
import Data.Typeable
import Data.IntMap (IntMap)
import qualified Data.IntMap as IntMap
data Dispatcher =
  D (IntMap Dynamic) (Maybe (Dynamic → IO ()))
empty :: Dispatcher
empty = D IntMap.empty Nothing
fallback :: Maybe (Dynamic → IO ())
       → Dispatcher → IO Dispatcher
fallback f (D d _) = return (D d f)
insert :: (Typeable a) ⇒ a → (a → IO ())
       → Dispatcher → IO Dispatcher
insert t m (D d f) = do
  k ← typeRepKey (typeOf t)
  return (D (IntMap.insert k (toDyn m) d) f)
delete :: (Typeable a) ⇒ a → Dispatcher
       → IO Dispatcher
delete t (D d f) = do
  k ← typeRepKey (typeOf t)
  return (D (IntMap.delete k d) f)
dispatch :: Dispatcher → Dynamic → IO ()
dispatch (D methods fallback) dyn = do
  key ← typeRepKey (dynTypeRep dyn)
  case (IntMap.lookup key methods) of
    Just method → do
      case (dynApply method dyn) of
        Just r → case (fromDynamic r) of
          Just r' → r'
          Nothing → error "never reached"
        Nothing → error "never reached"
    Nothing → case fallback of
      Just f → f dyn
      Nothing → putStrLn "no method"
```

```
module DispatcherTest (test) where
import Dispatcher
import Data.Dynamic (toDyn)
test = let f s = putStrLn ∘ (s++) ∘ show
do
  return empty ≫
  insert (⊥ :: Integer) (f "Integer: ") ≫
  insert (⊥ :: String) (f "String: ") ≫
  insert (⊥ :: [Int]) (f "[Int]: ") ≫ λd →
  dispatch d (toDyn (1 :: Int)) ≫
  dispatch d (toDyn "hello") ≫
  dispatch d (toDyn (1 :: Integer)) ≫
  dispatch d (toDyn ([0..9] :: [Int])) ≫
  delete (⊥ :: String) d ≫
  fallback (Just (f "Dynamic: ")) ≫ λd' →
  dispatch d' (toDyn "Hi!")
```

Figure 4: A *Dynamic* method dispatcher.

to retrieve it from the store. The obvious candidate is a symbol atom sufficiently unlikely to be used by any other code. Storing this symbol in non-Haskell-aware objects would lead to undefined behaviour if later sent to Haskell-aware objects, but there would be neither memory leak nor crash, which is an acceptable compromise.

Pd's depth first message passing makes a stack a useable data structure, but Haskell's strong static typing causes difficulties in a dynamic environment such as Pd⁵. Luckily there is a solution, in the form of the *Data.Dynamic* module. Values can be converted to *Dynamic* values, which contain run-time type information. These *Dynamic* values can be manipulated safely⁶ according to their "real" type (Figure 4).

The solution with *Data.Dynamic* is reasonably satisfactory, however this loses polymorphism; it would not be possible to have a Pd object that performs the Haskell operation `reverse :: [a] -> [a]`, instead a monomorphic type such as `reverse :: [Float] -> [Float]` would be required. In fact, this problem is non-trivial, and possibly impossible to solve without extending the compiler [8].

One approach could be to store source code for each object instead of compiling at class load time. Then, incoming messages from non-Haskell Pd objects would trigger a graph discovery algorithm, to find out which Haskell Pd objects are affected. Finally, the stored source code for these affected objects could be combined into an appropriate form to be compiled or interpreted as a single unit [9]. However, the effort would probably best be spent elsewhere.

8. HASKELL REALLY IN PD

Loading Haskell code from external files is the main purpose of `hsex`, but it would also be useful to be able to write Haskell functions inside Pd object boxes (similar to the expressions that can be written inside `[expr]` and `on`). The restrictions detailed earlier in this paper concerning *Dynamic* values and polymorphism apply here, too, thus it would be necessary to declare a monomorphic type with each function. Moreover, as Haskell functions are generally curried and as Haskell functions are valid values, it is unclear how many inlets are necessary. For example:

```
(+) :: Float -> Float -> Float
```

could have two inlets, each accepting a *Float*, with the object's output being a *Float*. But, that type is identical (in Haskell) to:

```
(+) :: Float -> (Float -> Float)
```

which could have one inlet, taking a *Float*, with the object's output being a function *Float -> Float*.

One way of solving the issue would be to have the maximal number of inlets possible, but that disallows any returning of functions. An acceptable alternative would be to change the semantics of parentheses, and have pseudo-Haskell code. In fact, this is already necessary, as Pd treats specially several characters crucial to Haskell source code text⁷, for which re-

⁵Recall the familiar error messages from Pd: "error: float: no method for 'symbol'"

⁶For example, by forbidding numeric addition of two symbols, a meaningless operation [7].

⁷Namely:

```
\ { } , ; $
```

placements would have to be made in the process of building a source code string from a collection of Pd atoms (being the creation arguments).

Since a type signature needs to be explicit, and since Haskell prefixes type signatures by `::`, it is logical to name the class that allows Haskell code in object boxes `::`. Marshaling to and from Pd types should be as automatic as possible, for example the user should be able to create an object box like `[:: [Float] -> Float -> [Float]; flip (:)]` and have it work as intended. When presented with a candidate type of `[Float]`, the implementation should check that, first, we have a selector of `list`, or `float` (for single element lists), or `bang` (for empty lists), and second, that all the elements of the list are indeed *Floats*. If both of these conditions are met, the incoming array of atoms should be converted to a list. If a return type is not marshalable to Pd directly (for example, given an object box containing `[:: [Float] -> [Float] -> ([Float],[Float]); (,)]`, it should be converted to a *Dynamic* value and outlet as such, for further processing by other Haskell objects.

`[::]` could be implemented by generating a source code file with an appropriate setup resource, and loading it as `hsex` loads other source code files. The class name defined in the resource would have to be suitably unique. The `[::]` object would use Pd to create a hidden object of the new class, and forward all inlet messages to it, and catch all outlet messages and forward them to Pd. As the *PureData* API will be stored in a record, the `[::]` can pass the new class an appropriately modified copy.

9. REFERENCES

- [1] Haskell Weekly News: January 31, 2007: Quotes of the Week, 2007.
- [2] The On-Line Encyclopedia of Integer Sequences: Fibonacci Numbers.
- [3] John Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98-107, 1989.
- [4] Simon Marlow and Simon Peyton Jones. The New GHC/Hugs Runtime System.
- [5] Manuel Chakravarty [editor] et al. The Haskell 98 Foreign Function Interface 1.0: An Addendum to the Haskell 98 Report. 2003.
- [6] Thomas Grill. [PD-dev] [pure-data-Patches-1353230] class loader hook, 2005.
- [7] Mathieu Bouchard. [PF] Re: concat, 2007.
- [8] Stefan O'Rear. [Haskell-cafe] Re: embedding Haskell: problematic polymorphism, 2007.
- [9] Claus Reinke. [Haskell-cafe] Re: embedding Haskell: problematic polymorphism, 2007.

APPENDIX

A. A FUTURE PUREDATA API FOR HASKELL

The main `hsex` Haskell code will build a record containing functions partially applied to the global mutable state it requires. This record will be passed to each setup resource, which can store and call the functions within it as and when it requires.

module PureData where

```
-- types, where “...” is additional data reserved for hsexth internal use
data Symbol    = Symbol PdSymbol ...
data Pointer   = Pointer PdPointer ...
data Atom      = AtomFloat  Float
                | AtomSymbol Symbol
                | AtomPointer Pointer
data Class     = Class PdClass ...
data Object    = Object PdObject ...
data Inlet     = Inlet PdInlet ...
data Outlet    = Outlet PdOutlet ...
data Receive   = Receive PdReceive ...
data Clock    = Clock PdClock ...

-- the master record of Pd API functions
data PureData = Pd{
  bang      :: Symbol, -- pure data
  float     :: Symbol,
  symbol    :: Symbol,
  pointer   :: Symbol,
  list     :: Symbol,
  gensym   :: String → IO Symbol, -- data management
  peeksym  :: Symbol → IO String,
  newClass  :: Symbol → ((Symbol, [Atom]) → IO Object) → (Object → IO ()) → IO Class, -- class management
  addCreator :: Symbol → ((Symbol, [Atom]) → IO Object) → Class → IO (),
  newObject :: Class → IO Object, -- object management
  freeObject :: Object → IO (),
  newInlet  :: Object → IO Inlet, -- inlet management
  freeInlet :: Inlet → IO (),
  addBang   :: IO () → Inlet → IO Inlet, -- inlet methods
  addFloat  :: (Float → IO ()) → Inlet → IO Inlet,
  addSymbol :: (Symbol → IO ()) → Inlet → IO Inlet,
  addPointer :: (Pointer → IO ()) → Inlet → IO Inlet,
  addList   :: ([Atom] → IO ()) → Inlet → IO Inlet,
  addMethod :: Symbol → ([Atom] → IO ()) → Inlet → IO Inlet,
  addAnything :: ((Symbol, [Atom]) → IO ()) → Inlet → IO Inlet,
  newOutlet :: Object → IO Outlet, -- outlet management
  freeOutlet :: Outlet → IO (),
  outBang   :: Outlet → IO (), -- outlet methods
  outFloat  :: Outlet → Float → IO (),
  outSymbol :: Outlet → Symbol → IO (),
  outPointer :: Outlet → Pointer → IO (),
  outList   :: Outlet → [Atom] → IO (),
  outAnything :: Outlet → (Symbol, [Atom]) → IO (),
  newReceive :: Symbol → IO Receive, -- receive management
  freeReceive :: Receive → IO (),
  rcvBang   :: IO () → Receive → IO Receive, -- receive methods
  rcvFloat  :: (Float → IO ()) → Receive → IO Receive,
  rcvSymbol :: (Symbol → IO ()) → Receive → IO Receive,
  rcvPointer :: (Pointer → IO ()) → Receive → IO Receive,
  rcvList   :: ([Atom] → IO ()) → Receive → IO Receive,
  rcvMethod :: Symbol → ([Atom] → IO ()) → Receive → IO Receive,
  rcvAnything :: ((Symbol, [Atom]) → IO ()) → Receive → IO Receive,
  newClock  :: IO () → IO Clock, -- clock management
  freeClock :: Clock → IO (),
  clockDelay :: Double → Clock → IO Clock, -- clock manipulation
  clockClear :: Clock → IO Clock,
  time      :: IO Double,
  timeSince :: Double → IO Double
} -- data PureData
```

Figure 5: Sketch of a future PureData API for Haskell.