Animated Pictures for Slide Presentations: From the Shallows to the Depths of a Domain-Specific Language (Functional Pearl)

OLIVER FLATT, University of Washington, USA
ROBERT BRUCE FINDLER, Northwestern University, USA
MATTHEW FLATT, University of Utah, USA

Another DSL for pictures? Seems fishy. But hold fast as we chart a course to an embedded DSL for the domain of slide presentations with animations. Our DSL programs interact with the host language in two ways: by allowing pictures and animations to be built using host-language functions (resembling a shallow embedding), and by allowing host-language reflection on their construction (resembling a deep embedding). As a result, users can define their own animation combinators but still also inspect, adjust, and reassemble animations. To demonstrate our DSL's expressive power, we show how it supports a Keynote-like Magic Move operation.

Having made the DSL shipshape, we dive into a lesson learned about deep and shallow DSL design. While deep and shallow may seem like mutually exclusive options, we argue they are actually points on an entire spectrum of possible DSL designs. More importantly, our paper demonstrates that points on the spectrum between deep and shallow are where DSL designs achieve full sail.

CCS Concepts: • **Software and its engineering → Domain specific languages**.

Additional Key Words and Phrases: shallow embedding, deep embedding, fish

ACM Reference Format:
Oliver Flatt, Robert Bruce Findler, and Matthew Flatt. 2026. Animated Pictures for Slide Presentations: From the Shallows to the Depths of a Domain-Specific Language (Functional Pearl). *Proc. ACM Program. Lang.* 10, ICFP, Article 294 (August 2026), 30 pages. https://doi.org/10.1145/3828692

1 Fish

Suppose that we're teaching functional programming by using a picture library, and we want to show students how the expression on the left produces the result picture on the right:

```
overlay(ellipse(~width: 50,
                ~height: 25,
                ~fill: "lightblue"),
        ~dx: 30, ~dy: 0,
        triangle(~size: 25,
                 ~fill: "lightblue")
         .rotate(1/2 * math.pi))
```

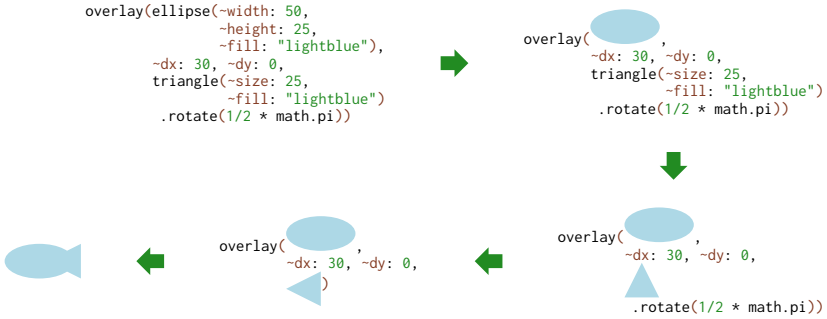

Let's assume that students are already familiar with the basic syntax of Rhombus (Flatt et al. 2023), so they can see that `overlay`, `ellipse`, `triangle` are functions that are called; that tilde-prefixed

Authors' Contact Information: Oliver Flatt, University of Washington, Seattle, WA, USA, oflatt@cs.washington.edu; Robert Bruce Findler, Northwestern University, Evanston, IL, USA, robby@cs.northwestern.edu; Matthew Flatt, University of Utah, Salt Lake City, UT, USA, mflatt@cs.utah.edu.

ACM 2475-1421/2026/8-ART294
https://doi.org/10.1145/3828692

names like `~width`, `~height`, and `~dx` are used for keyword arguments that are sometimes mixed among by-position arguments; and that `rotate` is a method of the object produced by `triangle`.

Naturally, we will show students how the program produces its result through algebraic reduction steps. As a static diagram, we might draw the evaluation sequence like this:



In class, however, each step between arrows should be a separate slide. Even better, instead of abruptly changing from one state to the next, the redex in each case should morph into its value. That animation deserves a live example that we can't include in a paper (in the literal sense), but we can plot a timeline showing snapshots from t=0 to t=1, where t=0 is the point where we advance a slide, and t=1 is the point where the next slide is in place:

Even this morphing animation may be too abrupt if a student isn't clear about which part they should be watching. So, before morphing each redex, we'd like to wiggle it a little bit, with a slight rotation back and forth:

The overall slide sequence is then: initial expression, (click to advance the slide) wiggle first redex, (click) morph redex to value, (click) wiggle second redex, (click) reduce second redex, and so on, until the overall expression is reduced to the fish result value.

2 Charting a Course

Depending on how you normally construct slides for presentations, you may start to imagine the kinds of "wiggle" and "magic move" operations that a WYSIWYG presentation application provides and how you would use them to build a fish animation. But we're hooked on functional programming, and we prefer to create slides through an embedded domain-specific language (DSL). As you have probably guessed, the `overlay`, `ellipse`, and `triangle` functions in the fish example

are also parts of the picture library that we will use to implement the slides, which we call `Pict`. We have much more to explain to reach animations.

Describing `Pict`, meanwhile, provides an opportunity to discuss the nature of embedded DSL design, and especially the choice between *shallow* versus *deep* embeddings. That choice is typically presented as a binary one, but our experience suggests that DSL design is not so neatly categorized. In this section we make ready by first reviewing the existing understanding of deep and shallow embeddings and giving a brief introduction to Rhombus.

With those preliminaries in our wake, section 3 shows where existing picture combinator libraries fall short, and it explains the new concepts in `Pict` to address those shortcomings. Section 4 explains the implementation of `Pict`, particularly noting how *deep* and *shallow* techniques work together. Section 5 then reflects on that synthesis, including section 5.3's discussion of some prominent DSLs, and it covers related work.

2.1 Shallow versus Deep

Boulton et al. (1992) provide the first description of *shallow* and *deep* as characterizations of an embedded DSL's design and implementation. Follow-up work by Claessen (2001), Jovanovic et al. (2014), and Rompf et al. (2013) echo and refine the distinction:

- A *shallow* embedding is one that directly uses host-language constructs as the DSL's constructs. Values in the DSL are represented by values in the host language, host-language functions are used to abstract over such values, and a host-language type system might ensure well-formedness of DSL terms.
- A *deep* embedding amounts to a DSL interpreter that is implemented within a host language, where host-language values represent the abstract syntax of a DSL program. Internally, the interpreter may use host-language values to represent DSL values, but that representation is not exposed. Functions in the host language can abstract over DSL abstract syntax, and thus only indirectly abstract over host values.

Gibbons and Wu (2014) (as anticipated by Reynolds (1975)), offer a crisp characterization based on the host type for a DSL program. In their formulation, the type of a deep DSL program is an ADT in the host language and the DSL program construction operators are simply constructors of the type, meaning that the interpreter for the DSL is a substantial recursive function. In contrast, the host type of a shallow DSL program generalizes the type of the DSL interpreter and the DSL program construction operators compose these functions together, meaning that the interpreter for the DSL is a trivial function that simply supplies initial values for the accumulator arguments.

There are advantages to both approaches; a deep DSL leaves open the possibility of new *interpretation* functions without needing to modify the DSL program ADT. As such functions can be used to optimize or otherwise analyze DSL programs after they have been constructed, deep designs tend to favor introspection. A shallow DSL leaves open the possibility of new *operations* in the DSL without modifying the type of DSL programs. As such operations tend to manifest as new ways to compose existing DSL programs, shallow designs tend to favor compositionality. As Gibbons and Wu (2014) point out, these tradeoffs parallel the tradeoffs in those of the extensibility/expression problem (Cook 1990; Flatt and Findler 1998; Krishnamurthi et al. 1998), specifically for DSL design.

2.2 Rhombus Crash Course

Rhombus (Flatt et al. 2023) fuses Python-like, whitespace-sensitive notation with Lisp-style macro extensibility. Rhombus's syntax is largely conventional and uses familiar notation for arithmetic, function calls, method calls, field access, indexing, and lists. To introduce specific Rhombus constructs that we use in the paper, consider these two functions:

```
fun sea_dragon(n :: Nat)
  :: List.of(Int.in(0,1)):
  match n
  | 0: []
  | ~else:
      def [x, ...] = sea_dragon(n-1)
      def [y, ...] = [x, ...].reverse()
      [x, ..., 0, 1-y, ...]
```

```
fun check_interleave(n :: Nat)
  :: Boolean:
  def [dn, dn1]:
    [sea_dragon(n), sea_dragon(n+1)]
  for all:
    each i in 0..dn.length()
    dn1[i*2+1] == dn[i]
      && dn1[i*2] == i mod 2
```

The `fun` form on the left defines `sea_dragon`, which accepts a natural number `n` and computes the steps needed to draw the `n`-th iteration of the Heighway dragon. Each declaration following `::` is a type-like annotation that is enforced via dynamic checking, whether `:: Nat` for the function argument or `:: List.of(Int.in(0, 1))` for the function result. The result annotation ensures that the result list contains integers that are all either `0` or `1`.

The body of `sea_dragon` contains `match` cases, each starting with `|`. If `n` is zero, the first case returns the empty list. If `n` is not zero, the `~else` case uses `def` to introduce local identifiers. These `def` forms consist of a binding and an expression on the same line, separated by a `=`.

Every binding position in Rhombus supports binding patterns, including binding using `def`, binding arguments using `fun`, and binding in the case of a `match`. The `def`s within `sea_dragon` use the binding *patterns* `[x, ...]` and `[y, ...]` to bind `x` and `y` to the elements of the corresponding lists. The `...` operator can also appear in list *expressions*, where it expands a variable that was bound by an ellipsis pattern back into its sequence of elements. The expression `[x, ...]` rebuilds the list that the pattern `[x, ...]` destructured. The expression `[x, ..., 0, 1-y, ...]` concatenates the `x` list elements with the singleton list containing `0` and the list obtained by mapping the function `fun (y :: Int): 1-y` over the `y` list elements.

The `.` operator is overloaded. Some uses access an object's field or method, as in the method call `[x, ...].reverse()`. Other uses build a hierarchical name, as in `List.of` and `Int.in`.

The second example function, `check_interleave`, checks a property of the Heighway dragon, namely that one iteration of the curve can be obtained from the previous one by interleaving 0s and 1s into it. It uses an alternative form of `def` with a `:` in place of `=`. Using `:` allows the right-hand side to appear on an indented new line.

The `check_interleave` function also demonstrates Rhombus's `for` loops. Each `for` loop starts with a *reducer* that says how successive results of the body are combined. The `check_interleave` loop uses the `all` reducer, which stops the `for` loop and returns false if any iteration produces false, otherwise it produces the result of the last iteration. Other options include `any`, which terminates the loop as soon as an iteration returns a true value, and the `fold` reducer, which binds a variable to an initial value or the result of each previous iteration of the body. A common pattern with `fold` is to duplicate a variable, e.g., `for fold(p = p)` naming the accumulator `p` in the body and initializing it to the value of `p` in the scope outside the loop.

An `each` form within a `for` loop binds an induction variable to the elements of an iterable value. In this case, the variable `i` takes on the integers from `0` up to one fewer than the length of `dn`. Since the position after `each` is a binding position, `i` could have been a pattern matched against each element of the iterable value.

The `sea_dragon` function returns a boring list of numbers, because we have not yet explained the `Pict` library for producing pictures. See the appendix for a function that turns a dragon list into a picture.

3 Diving into the Problem

We set sail with a picture combinator library that resembles many others, such as the libraries described by Henderson (1982), Finne and Peyton Jones (1995), and Findler and Flatt (2004). These libraries take care of the details of composing picture elements and eventually rendering them, but they rely on the host language to let programmers channel elements of a scene, such as mapping over a list of pictures to scale each of them. Indeed, the kinds of animations that we're aiming for *could* be implemented over such a picture library by taking advantage of host-language abstractions, such as writing a function that takes the time t as an argument. Our experience suggests that animations *shouldn't* be implemented that way, however. Instead, more should be added to the picture library itself, expanding the domain to cover animations.

3.1 Swimming in the Shallows

The `overlay`, `ellipse`, and `triangle` functions on the example slides look like a *shallow* embedding of a DSL. In a shallow embedding, domain constructs are provided as functions and methods, and calling a function or method produces the result picture. Here are some more examples evaluated at the Rhombus read-eval-print loop (REPL):

```
> circle(~size: 10, ~fill: "blue")
```

```
> triangle(~width: 10, ~line: "blue")
```

```
> triangle(~size: 10).colorize("red")
```

```
> beside(circle(~fill: "blue", ~size: 10),
         triangle(~line: "blue", ~size: 10)).scale(3)
```

Each call to the functions `circle`, `triangle`, `ellipse`, and others like them create a `Pict` object that has a default size, draws an outline with a pen using an inherited color by default, and does not fill by default—or, equivalently, fills with a transparent brush. Keyword-based arguments can override the default size, pen color, brush color, and other attributes of the drawing. Functions like `beside`, `stack`, and `overlay` combine picts by arranging them horizontally, vertically, or on top of each other, producing a pict as the combination. Methods on a pict like `scale` and `colorize` create a new pict that is scaled or that replaces an inherited color with the given one.

Picts can contain text too; here are some examples:

```
> text("hello").colorize("forestgreen")
```

```
> stack(text("shadow"),
        text("shadow").vflip().shear(0.5,0).translate(2,-4).colorize("gray"))
```

```
> rhombusblock(1 + 2).hflip()
```

The `text` function straightforwardly renders a string as a pict drawing the string's text. The `rhombusblock` form, however, is not a function. It's a macro that inspects the syntax of its argument and constructs a Pict that renders as the source term. That's a Rhombus-specific example of taking

advantage of the host language in an embedded DSL, where the macro system makes a source-rendering syntactic form possible.

A more typical use of the host language is to parameterize a picture construction by defining a function. For example, we can define a `make_overlay` function that takes picts for subexpressions of an `overlay` call. This function takes advantage of `rhombusblock`'s support for the escape operator `#,`. The escape operator places its argument into the enclosing `rhombusblock` picture in a manner similar to string interpolation, but it places pictures inside pictures, instead of strings inside of strings. In this case, the `sub1` argument to `make_overlay` will appear as the argument to `overlay` in the resulting pict. Accordingly, we can call the function with either picts that show code or picts that show the result of the code, preparing the way for our animation.

```
fun make_overlay(sub1 :: Pict, sub2 :: Pict) :: Pict:
  rhombusblock(overlay(#,(sub1),
                       ~dx: 30, ~dy: 0,
                       #,(sub2)))

> make_overlay(rhombusblock(ellipse(~width: 50, ~height: 25)),
               rhombusblock(triangle(~width: 25)))
overlay(ellipse(~width: 50, ~height: 25),
        ~dx: 30, ~dy: 0,
        triangle(~width: 25))

> make_overlay(ellipse(~width: 50, ~height: 25),
               triangle(~width: 25))
overlay(
```

```
        ~dx: 30, ~dy: 0,
```

```
        )
```

Along the same lines, we can set up the animation from an expression to its value by parameterizing a combination over a number from `0` (fully expression) to `1` (fully value), fading from one to the other by alpha compositing:

```
fun make_morph(n :: Real.in(0, 1), from :: Pict, to :: Pict) :: Pict:
  overlay(from.alpha(1 - n), to.alpha(n))

> make_morph(1/3, rhombusblock(triangle(....)), triangle(~width: 25))
triangle(....)
> make_morph(2/3, rhombusblock(triangle(....)), triangle(~width: 25))
triangle(....)
```

3.2 Against the Tide

The preceding section seems to be on the right tack for generating our example slide sequence, because the evaluation sequence will need to construct picts that replace specific subexpressions with their values as well as the intermediate morphs. It turns out that representing animations as parameterized functions in the host language makes larger animations surprisingly difficult to construct. To make this more concrete, let us first define a few pictures for some expressions and values that we will need. We adopt a naming convention used throughout the rest of the paper: a

`_code` suffix names a pict that shows source code (built with `rhombusblock`), while a `_val` suffix names a pict that shows the value that the code produces.

```
def ellipse_code = rhombusblock(ellipse(~width: 20, ~height: 10))

def ellipse_val = ellipse(~width: 20, ~height: 10)

def tail_code = rhombusblock(triangle(~size: 10).rotate(1/2 * math.pi))

fun make_beside(sub1 :: Pict, sub2 :: Pict) :: Pict:
  rhombusblock(beside(#,(sub1),
                      #,(sub2)))
```

Here's a sketch of some early steps:

```
> make_beside(ellipse_code, tail_code)
beside(ellipse(~width: 20, ~height: 10),
       triangle(~size: 10).rotate(1/2 * math.pi))

> make_beside(make_morph(0.25, ellipse_code, ellipse_val), tail_code)
beside(ellipse(~width: 20, ~height: 10),
       triangle(~size: 10).rotate(1/2 * math.pi))
> make_beside(ellipse_val, tail_code)
beside(⬭,
       triangle(~size: 10).rotate(1/2 * math.pi))
```

This code is calling `make_morph` just once to illustrate the idea, but for the full animation, we'll need to call it several times with values between 0 and 1. More problematically, however, the next evaluation step happens inside `tail_code`, so we must go back and parameterize that subexpression's construction:

```
fun make_tail_code(triangle :: Pict) :: Pict:
  rhombusblock(#,(triangle).rotate(1/2 * math.pi))

def triangle_code = rhombusblock(triangle(~size: 10))
def triangle_val = triangle(~size: 10)

> make_beside(ellipse_val, make_tail_code(triangle_code))
beside(⬭,
       triangle(~size: 10).rotate(1/2 * math.pi))
> make_beside(ellipse_val,
              make_tail_code(make_morph(0.5, triangle_code,
                                        triangle_val)))
beside(⬭,
       triangle(~size: 10).rotate(1/2 * math.pi))
> make_beside(ellipse_val, make_tail_code(triangle_val))
beside(⬭,
       △.rotate(1/2 * math.pi))
```

This is starting to get tedious, and it begs for automation to deal with any kind of expression and its nested subexpressions—but belay that thought until section 3.4. The specific nesting of this case is representative of simple animations that appear frequently in slide presentations. In such simple animations, more general automation is not needed, but implementing an animation effect this way is still too much code written outside of the picture DSL.

Fundamentally, the problem is that the representation of animations as parameterized functions in the host language isn't *composable*. We must manually compute the local time that `make_morph` expects based on the global time that the entire animation is running. Even worse, we end up writing the entire animation inside-out, as an abstraction of an enclosing expression that is wrapped around suitable subexpressions. Each time we want to change a subexpression, the entire surrounding context is either duplicated or abstracted into a parameterized function. To put it another way, we are no longer working with pictures in the DSL, but with host-language constructs representing values that encode a new idea. We end up having to manually lift all of our pict operations to a parameterized-function layer. At the very least, we need a new DSL for that layer.

Should that DSL be a new one that floats on top of a pict DSL, or should we expand the pict DSL to accommodate animation? Much of the turbulence in this example stems from the inability of pict combinators to act on animations, leading us to expand the pict DSL with an `animate` constructor, instead of replicating its functionality in a new layer. It will wrap a time-parameterized picture-generating function as an animated pict, which can then be used with all the other pict operations:

```
def ellipse_anim:
  animate(
    fun (n :: Real.in(0, 1)):
      make_beside(make_morph(n, ellipse_code, ellipse_val),
                  tail_code)
  )
```

Examining the animation directly in the REPL produces t=0 snapshot of the animated picture:

```
> ellipse_anim
beside(ellipse(~width: 20, ~height: 10),
       triangle(~size: 10).rotate(1/2 * math.pi))
```

The `explain_anim` library function shows multiple snapshots of a pict, so we can see, on a static page, more of what the animation will show:

```
> explain_anim(ellipse_anim).scale(0.3)
```

3.3 The Deep End

Internalizing the idea of animation gets us off on a good heading out of these choppy waters. It is not smooth sailing yet, however, so instead of the inside-out representation that host-language functions encourage, let's first just give names to subexpression picts as we build up the overall expression directly:

```
def triangle_code = rhombusblock(triangle(~size: 10))
def tail_code = rhombusblock(#,(triangle_code).rotate(1/2 * math.pi))
def fish_code = rhombusblock(beside(#,(ellipse_code),
                                    #,(tail_code)))

def tail_val = triangle_val.rotate(1/2 * math.pi)
```

Then, we would like to be able to *replace* an expression picture with its value picture:

```
> fish_code
beside(ellipse(~width: 20, ~height: 10),
       triangle(~size: 10).rotate(1/2 * math.pi))
```

```
def step1 = fish_code.replace(ellipse_code, ellipse_val)

> step1
beside(⬭,
       triangle(~size: 10).rotate(1/2 * math.pi))

def step2 = step1.replace(triangle_code, triangle_val)

> step2
beside(⬭,
       △.rotate(1/2 * math.pi))

def step3 = step2.replace(tail_code, tail_val)

> step3
beside(⬭,
       ◁)
```

With the addition of `replace`, we should pause to take our bearings, as it is quite different from any `Pict` primitives we've defined so far. While other operations need only observe the metadata of children picts, `replace` needs to observe and transform the deep structure of the pict itself. In fact, we would like `replace` to replay the way it was constructed after the replacement. Specifically, notice how the commas and closing parentheses correctly appear immediately after a replacement picture, instead of being stuck to the locations that they assumed around the original subexpression pictures. This means `replace` is necessarily a program transformation over the pict DSL, and not just an operation on the final pict value that the DSL produced. Supporting `replace` suggests that we will need a *deep* DSL approach.

Replacing a static pict with another static pict is a step forward, but for our target fish animation, we want to be able to replace a static expression pict with an animation, either a wiggling expression or an expression morphing to its value.

```
fun wiggle(p :: Pict) :: Pict:
  animate(
    fun (n :: Real.in(0, 1)):
      let q = p.rotate(1/10 * math.sin(n * 2 * math.pi))
      overlay(p.ghost(), q.launder()).refocus(p),
  )
```

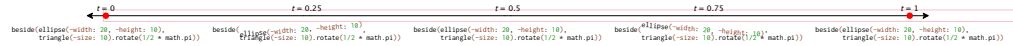
Replacing a static pict with an animated one makes the enclosing pict animated:

```
> explain_anim(fish_code.replace(ellipse_code, wiggle(ellipse_code)))
    .scale(0.3)
```

Finally, we need to sequence animations. To do so, our picture DSL offers the `switch` constructor, which is similar to `beside`, but in the time dimension instead of the horizontal dimension. Operations like `switch` help explain why the above two timelines stop just before t=1, namely to avoid an overlap with the next animation's pict at t=0. To stretch an animation to t+1, we will need a `sustain` operation. To see how it works, we use `explain_anim`'s `~start` and `~end` keyword arguments that show us specific intervals of t.

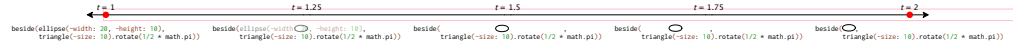
```
def steps:
  switch(fish_code.replace(ellipse_code, wiggle(ellipse_code)),
         fish_code.replace(ellipse_code, morph(ellipse_code, ellipse_val)))
```

```
> explain_anim(steps, ~start: 0, ~end: 1).scale(0.3)
```




```
> explain_anim(steps, ~start: 1, ~end: 2).scale(0.3)
```


```
> explain_anim(steps.sustain(), ~start: 1, ~end: 2).scale(0.3)
```



Each increment of t by 1 corresponds to a slide that a presenter steps forward by clicking. The `duration` property of a pict reports how long its animation lasts in such time steps, which are called *epochs*. The time in seconds taken to animate an epoch (before waiting for the click to advance) is its *extent*, and `animate` uses a 0.5-second extent by default.

```
> steps.duration
2
> steps.sustain().duration
3
> steps.epoch_extent(0)
0.5
> ellipse_code.epoch_extent(0) // non-animated pict
0
```

3.4 Off the Deep End: Automating Evaluation

In the example scenario of teaching functional programming, we will want to show students several examples of evaluation, not just the one to create a fish. To automate multiple examples, we'll want tools to take an example expression, like the one to create a fish, and extract expression pictures, result values, and an evaluation schedule.

In languages without sophisticated metaprogramming, implementing this automation without extending the DSL would be sailing upwind, but Rhombus's macro system allows us to stay comfortably in the host language, creating no wake. Specifically, we can write an `eval_steps` syntactic form as a macro that takes an expression and instruments its evaluation to produce a pict for the overall expression source and a list of evaluation steps, where each step is itself a 2-element list containing a *from* subexpression pict and a *to* value pict.

```
def [fish_code :: Pict, fish_steps :: List.of(List.of(Pict))]:
  eval_steps:
    overlay(ellipse(~width: 50,
                    ~height: 25,
                    ~fill: "lightblue"),
            ~dx: 30, ~dy: 0,
            triangle(~size: 25,
                     ~fill: "lightblue")
              .rotate(1/2 * math.pi))
```

On the second line, `eval_steps:` is the macro form, and the rest of the code is its body, the expression to instrument. The macro produces a 2-element list, which is then destructured by the binding pattern `[fish_code, fish_steps]` to bind `fish_code` to the source pict and `fish_steps`

to the list of steps. The implementation of `eval_steps` is an interesting exercise in using Rhombus binding spaces (Flatt et al. 2023) to give two different meanings to the body of an `eval_steps` form; roughly, the presence of `eval_steps` allows us to define different versions of the constructs used in its body that can simultaneously construct a picture and also record the steps of its evaluation, using Rhombus's macro technology. Importantly, such sophistication *doesn't* need to be a built-in part of the `Pict` DSL. Here, unlike the earlier attempt to define animations as host functions from time to picts, we've found a place where we really should just use the host language, relying on pict to provide a good representation for the result of `eval_steps` as a starting picture, ending picture, and intermediate evaluation steps.

Along similar lines, there are plenty of situations where everyday functional abstraction over a pict construction is both the right and convenient choice. Building every kind of possible abstraction into the `Pict` datatype is clearly not the right idea. The art of designing a DSL like `Pict` is in finding the concepts that need to be built into the abstraction and leaving out the rest.

3.5 Reel it Back In: Completing the Animation

We don't yet have all the pieces we need to build slides, but we have the essential ideas, and we can complete the animation code. This subsection shows the code from the top down.

To turn `fish_code` plus `fish_steps` into an animation, we write a `small_steps` function:

```
def fish_eval_steps = small_steps(fish_code, fish_steps)
```

The `small_steps` (plural) function is just a fold over a `small_step` (singular) function:

```
fun small_steps(p :: Pict, steps :: List) :: Pict:
  for fold (p = p):
    each [from, to] in steps
    small_step(p, from, to)
```

The `small_step` function uses more `Pict` library facilities that we will explain in the rest of the paper: `snapshot` to extract a static pict from an animated pict (in this case, the end of the animation so far) and `find_rebuilt` to find a subpict even if it has been modified (in this case, because the expression might have evaluated subexpressions replaced by values):

```
fun small_step(prev_p :: Pict, from :: Pict, to :: Pict) :: Pict:
  def p = prev_p.snapshot(prev_p.duration-1, 1)
  def from = p.find_rebuilt(from)
  def new_p = wiggle_then_morph(p, from, to)
  switch(prev_p, new_p, ~join: SequentialJoin.splice)
```

The `wiggle_then_morph` function produces an animated picture by replacing the `from` pict with a wiggling version of the `from` pict, then it separately replaces `from` with a pict that morphs from `from` to `to`, and then it puts those two resulting animations in sequence:

```
fun wiggle_then_morph(p :: Pict, from :: Pict, to :: Pict) :: Pict:
  def w_p = p.replace(from, wiggle(from))
  def m_p = p.replace(from, morph(from, to))
  switch(w_p, m_p).sustain()
```

The `morph` function is provided by the Rhombus `Pict` library, but it can be defined in terms of the primitives that we will describe. Its implementation is similar to `animate` plus `make_morph` from before, but it also morphs the graphical dimensions of the start and ending images, instead of just using the maximum bounds that `overlay` produces.

Except for `eval_tree`, `morph`, and the primitives to be described, that's the complete implementation of the animation—evidence that we're fishing in the right DSL pond.

3.6 Riding a Wave to Shore: Emulating Keynote's Magic Move

For those who use WYSIWYG slide-presentation software, one of the most popular features for animation is the Magic Move operation in Apple's Keynote. Typically, a slide is duplicated, and then objects are modified, added, or removed. A Magic Move transition from the first slide to the second will fade out removed objects, fade in added objects, and adapt modified objects by interpolating movement, scaling, and recoloring. If the second slide in a transition does not start as a duplicate of the first, Keynote's animation will infer a relationship between objects, but the transition is mainly intended to be used via slide duplication.

The idea behind Magic Move comes from the commonly used idea in animation of a keyframe, where the slides before and after serve as key frames for an animation. The idea adapts well to our setting as long as we can identify matching picts in the two slides. This boils down to defining an equivalence relation on picts. We return to the definition of that relation in section 4.3; for now we simply note that it is the same relation that `replace` uses to find matching picts.

In more detail, the `magic_move` function is implemented by

- traversing the dependencies from the *from* and *to* picts to determine which dependencies they have in common;
- for each dependency pict in common, it is replaced in *from* with an animated pict that drifts towards its position and scale in *to*;
- similarly, each common pict is replaced in *to* with an animated pict that drifts from its position and scale in *from*; and
- the *from* and *to* picts are otherwise merged by the `morph` function we used before, so removed picts fade out and added picts fade in.

For example, suppose we have a `one_fish` function that builds the fish-construction expression, but abstracts over the size and the color of the fish. We can use Rhombus's ellipsis notation to collect a list of schools of fish together, binding `leader` to all of the leaders of the schools and `follower` to all of the followers in the schools:

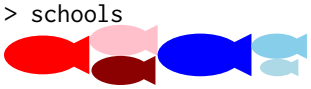
```
def [[leader, follower, ...], ...] = [[one_fish(40, "red"),
                                       one_fish(32, "pink"),
                                       one_fish(30, "darkred")],
                                      [one_fish(44, "blue"),
                                       one_fish(26, "skyblue"),
                                       one_fish(18, "lightblue")]]
```

Rhombus's ellipsis notation also enables a compact way to work with the leaders and followers. We can use it to define two picts, one that collects the schools together and one with the fish in a line, keeping the leaders up front.

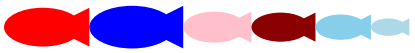
```
def schools = beside(beside(leader,
                            stack(follower, ...)),
                     ...)

def leaders = beside(leader, ...,
                     beside(follower, ...), ...)
```

```
> schools
```

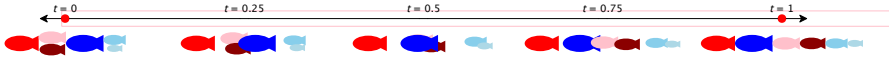


```
> leaders
```



The `magic_move` function can then interpolate between those two arrangements.

```
> explain_anim(magic_move(schools, leaders), ~end: 1).scale(0.4)
```



Of course, any animation that can be implemented using Magic Move could also be implemented by manually calculating the scale and position for each component as a function of time. In fact, when building slide presentations in earlier versions of our `Pict` DSL that did not support animations as a direct part of the library, we have (laboriously) done so. More important than the saved labor, however, using `magic_move` enables us to decompose parts of a slide presentation implementation more effectively, combining pieces into an animation after the fact.

We did not design `Pict` specifically to support Magic Move, so being able to implement `magic_move` as a library function is more evidence that we've landed the DSL that we were fishing for.

4 Set Sail Again: The Animated Picture Datatype

Now that we've shown a number of examples to illustrate the *why* of various pict properties, let's cast off for a second time, explaining *how* the `Pict` abstraction works. We'll start with a simplified definition and gradually add in the pieces to build up the full Rhombus `Pict` abstraction. The point of this section is *not* to explain the full `Pict` library, but to show enough of the `Pict` implementation to chart its specific course through the *shallow* and *deep* waters of DSL design.

Looking ahead, two features of `Pict` combine to give the DSL its expressive power. First, users can write new animation combinators like `wiggle`, extending the DSL with new operations. Supporting this capability suggests a shallow DSL design, since `wiggle` is written as a host-language function that takes a `Pict` as an argument and produces a new `Pict` as a result. Second, users can write new operations like `magic_move`, which needs to inspect and modify the way a `Pict` is constructed. Supporting this capability suggests a deep DSL design, since `magic_move` traverses and adjusts a `Pict`'s construction. These two capabilities may seem incompatible, but they correspond to the shallow and deep ends of DSL design, respectively. Our `Pict` abstraction is a hybrid that supports both simultaneously. Indeed, with a small tweak to the `wiggle` implementation we can even support replacement of the `Pict` that `wiggle` wiggles.

We start back in the shallows with static and animated pictures.

4.1 Drawing and Bounding Boxes

A static pict, as found in many functional picture libraries, can be defined as a Rhombus record (using the `class` form, whose notation is similar to Scala's `class` form) with three fields: a width, a height, and a drawing function that takes a target surface and offset.

```
class Pict(width :: Real,
           height :: Real,
           draw :: (Surface, Real, Real) -> Void,
           ....) // additional fields elided, for now
```

The `Pict` class declaration above is meant to explain its properties, but users of the library cannot construct a `Pict` directly, and they instead use functions like `rectangle` and `beside` to create and

compose `Pict` objects. For example, the `rectangle` function produces a `Pict` with exactly the given `width` and `height`. Its `draw` function draws a rectangle of that size on the target `Surface`.[1]

Here's a simplified `beside` function that composes two picts by adding their `width` values, taking the maximum of the `height` values, and creating a new `draw` function that calls the given picts' `draw` functions:

```
fun beside(left :: Pict, right :: Pict) :: Pict:
  // top-aligning variant of `beside`:
  Pict(left.width + right.width,
       math.max(left.height, right.height),
       fun (surface, dx, dy):
         left.draw(surface, dx, dy)
         right.draw(surface, dx + left.width, dy))
```

The `width` and `height` of a `Pict` are its *bounding box*, but "bounding" is not meant to insist that all drawing occurs within the specified dimensions. Instead, the bounding box is used as a guide for relative alignment, as in `beside` above, and it is sometimes useful to have a pict's rendering extend beyond the bounding box that is used for relative alignment. For example, starting with a concatenation of shapes

and highlighting the middle shape

is most easily implemented by replacing the middle shape with one that has the same bounding box but draws an extended border around the shape. Preserving the bounding box maintains the spacing between the shapes while the middle one is highlighted.

Sometimes, it's useful to create a pict that has the same bounds as another, but whose drawing is empty. The `Pict.ghost` method of our simplified `Pict` form returns such a pict:

```
class Pict(....): // the same fields as above, elided here
  method Pict.ghost() :: Pict:
    Pict(width, height, fun (surface, dx, dy): #void)
```

Other useful bounding-box operators on picts include the `Pict.pad` method, which adds space around a pict by extending its bounding box, and `Pict.clip`, which adjusts a pict's drawing so that the ink outside of the bounding box is discarded.

To better support text alignment, a Rhombus `Pict` includes an `ascent` field (distance from the top to the top baseline) and `descent` field (distance from the bottom to the bottom baseline) along with `width` and `height`. We leave those out here, because they do not require new techniques. Other graphical measurements could make sense as long as primitive combinators like `beside`, `stack`, and `overlay` can combine them.[2]

[1] The Rhombus `Pict` library provides a `dc` function that accepts an arbitrary drawing function using the Rhombus `draw` library's imperative API. In that library, `Surface` is actually called `DC`, which stands for "drawing context," but `Surface` seems clearer for our purposes here.

[2] One additional useful measurement is an "ink extent" covering the region where drawing actually occurs in a pict. That extent is available indirectly through `refocus_around_ink`, which we use in the appendix.

4.2 Animation

So far, our pict representation supports only static images. Generalizing to animations requires us to include a notion of time in the representation. As our library is designed to be used for slide presentations, a conventional notion of time like *seconds* is not appropriate. Instead, a moment in time consists of a natural number indicating the current slide, dubbed the *epoch*, and a number of seconds indicating how far we are in a transition to the next slide. The total number of seconds that each outgoing transition takes is called its *extent*, and the total number of epochs that a pict spans is called its *duration*. Conceptually, all picts exist in all epochs, much in the same way that all picts have the entire coordinate system at their disposal when drawing. The duration defines a time box, which plays a role similar to the bounding box. In the same way that most picts do not draw outside their bounding boxes, most picts do not draw in epochs beyond their duration. Still, the existence of the duration does not force the pict to draw only in those epochs. Instead, the duration exists to support composition, just like the bounding box.

We use `Time` values to capture a specific moment in an animation. Each `Time` contains an `Int` for the epoch and a `Real.in(0, 1)` indicating progression within the epoch during animation.

```
class Time(epoch :: Int,
           n :: Real.in(0, 1))
```

If a pict is used directly as slide presentation, epoch 0 corresponds to the first slide. The pict might instead be sequenced with another, in which case its epochs are effectively shifted. Although epochs are normally positive numbers, the process of building an animation may involve negative epochs to shift an animation back in time before combining it with another animation.

In general, some properties of picts need to become time-varying to support animation, including the pict's width, height, and drawing function. Accordingly, instead of having `width` and `height` fields, we have `width_at` and `height_at` fields as functions that depend on the time. As a convenience, we use `class`'s support for properties so that `p.width` and `p.height` report the width and height at the start of epoch 0 for any pict `p`. Also, the `duration` of a pict records the number of epochs it spans, and the `extent` method gives the extent in seconds for each epoch.

```
class Pict(width_at :: Time -> Real,
           height_at :: Time -> Real,
           draw_at :: Time -> (Surface, Real, Real) -> Void,
           duration :: NonnegInt,
           extent :: Int -> Real, ....): // yet more fields elided here
  property width: width_at(Time(0, 0))
  property height: height_at(Time(0, 0))
```

By default, shape primitives like `ellipse` create a pict with a duration of `1` epoch. Since the result of `ellipse` has no animation, its sole epoch has extent 0:

```
fun ellipse(~width: width, ~height: height, ....): // more arguments elided
  Pict(fun (t :: Time): width,  // constant width
       fun (t :: Time): height, // constant height
       fun (t :: Time):
         if t.epoch == 0
         | fun (surface, dx, dy): ....   // draw in epoch 0 (elided)
         | fun (surface, dx, dy): #void, // nothing otherwise
       1,                        // duration
       fun (epoch :: Int): 0)    // all epochs have extent 0
```

Animations can be created using the `animate` function, shown in the top half of figure 1. It creates an animated pict that samples its argument function, on demand, at offsets within epoch 0. Optionally, the `~extent` argument can specify the animation's extent in seconds. To create a multi-epoch animated pict, multiple calls to `animate` can be combined with `switch`. Internally, `switch` selects the correct dependency and computes a new relative `Time` using a helper function `at`. Its implementation is the bottom half of figure 1.[3]

4.2.1 Lowering Animation to Static Picts with Snapshots. To render an animated pict to a `Surface`, a slide-presentation layer samples the animation's drawing at various offsets within a slide's epoch. Each sample is a *snapshot* of the pict. The actual number of samples depends on the speed of the drawing context, but typically the animation is sampled around 20 times per second.

Beyond rendering a pict to a slide, snapshots are also useful to construct picts. For example, an animation might be composed from two existing animations by taking a snapshot at the end of the first animation and the start of the second animation, and then morphing between the two snapshots. As another example, the `explain_anim` function that we have been using to show animations throughout the paper is implemented by taking snapshots of the given pict.

The `Pict.snapshot` method takes the fields of a `Time` and returns a sample of an animated pict as a new pict. The result of `Pict.snapshot` is a `StaticPict`, which is still a `Pict`, but one that is guaranteed not to be animated. Functions like `triangle`, `ellipse`, and `text` also produce `StaticPict` results. We previously described the constraint on a `sample` function provided to `animate` as `Real.in(0, 1) -> Pict`, but it is more precisely `Real.in(0, 1) -> StaticPict`. That is, the result of `sample` is checked to ensure that a `StaticPict` is produced.

The fact that most `Pict` operations accept arbitrary `Pict`s and, generally speaking, do not require `StaticPict`s, is a central feature of the library. That feature is one of the reasons why we have been able to build up entire slide presentations from animated `Pict`s. Users can freely compose `Pict`s in space without considering time, compose `Pict`s in time without yet considering space, or write more complex compositions that are concerned with both space and time.

4.2.2 Snapshots and Lifting Static Composition to Animation. The `Pict.snapshot` method and `animate` function can be combined to lift any operation over `StaticPict`s to an operation over animated picts. For example, we may want to lift a function `static_morph` that interpolates between the bounding boxes of two picts. The natural lifting is to apply the morphing function pointwise in time, using snapshots to obtain static picts. A derived `morph` function that works with animated picts—interpolating bounding boxes pointwise in time—can be implemented as follows:

```
fun morph(from, to):
  // assuming `from` and `to` have a duration of 1
  animate(fun (n :: Real.in(0, 1)):
            static_morph(from.snapshot(0, n), from.snapshot(0, n)))
```

The `animate_map` function provided by the `Pict` library encapsulates this pattern to make lifting more explicit. It also handles multiple epochs by lazily mapping all epochs. The function passed to `animate_map` is guaranteed to get `StaticPict` arguments, and it must return a `StaticPict` result.

4.2.3 Concurrent and Sustained Animations. When using `animate_map` to generalize a function like `beside` to accept animated pict arguments, care must be taken, because the the input picts may have different durations, and they may have different extents within each epoch. The function

[3]The `Time` datatype is not part of the `Pict` external interface, which instead uses separate `epoch` and `n` arguments, but `Time` provides a clearer explanation of `Pict` internals.

```
fun animate(sample :: Real.in(0, 1) -> Pict,
            ~extent: extent :: Real = 0.5) :: Pict:
  Pict(fun (t :: Time): sample(clamp(t)).width,
       fun (t :: Time): sample(clamp(t)).height,
       fun (t :: Time): if t.epoch == 0
                        | sample(t.n).draw
                        | fun (surface, dx, dy): #void,
       1, // duration
       fun (epoch :: Int): extent)

// return `n` if `epoch` is 0, otherwise return 0 before and 1 after
fun clamp(Time(epoch, n)) :: Real.in(0, 1):
  cond
  | epoch < 0: 0
  | epoch == 1: n
  | epoch > 0: 1

fun switch(dep :: Pict, ...) :: Pict:
  Pict(fun (t :: Time):
         def [dep, dt] = at(t, [dep, ...])
         dep.width_at(dt),
       fun (t :: Time):
         def [dep, dt] = at(t, [dep, ...])
         dep.height_at(dt),
       fun (t :: Time):
         def [dep, dt] = at(t, [dep, ...])
         dep.draw_at(dt),
       math.sum(dep.duration, ...),
       fun (epoch :: Int):
         def [dep, dt] = at(Time(epoch, 0), [dep, ...])
         dep(dt.epoch))

fun at(t :: Time, deps :: NonemptyList.of(Pict)) :: values(Pict, Time):
  match deps
  | [dep]:
      [dep, t]
  | [dep0, _, ...] when t.epoch < dep0.duration:
      [dep0, t]
  | [dep0, dep, ...]:
      at(Time(t.epoch - dep0.duration, t.n), [dep, ...])
```

Fig. 1. Implementations of `animate` and `switch`

`concurrent` handles the job of normalizing the picts in the time dimension. It accepts a list of `Pict`s and returns a list of `Pict`s that all have the same duration and all have the same extent.

There are multiple ways that `concurrent` can handle extent adjustments. An animation might be slowed to a new extent, aligned to the beginning of a new extent (and thus completing earlier

than its nominal extent), or aligned to the end of a new extent (meaning it starts late), for example, depending on how two animations are meant to interact. An optional argument to `concurrent` selects how extents are combined, and the default is to center an animation within a longer extent.

Similar concerns apply to increasing a pict's duration, but `concurrent` applies a different default strategy for epochs: epoch 0 is aligned for all picts that are made concurrent, and the time box of each shorter-duration pict is extended by *sustaining*. Normally, a pict does occupy space (via its bounding box) outside of its duration, but it does not draw outside of its duration—so, sustaining a pict changes how it draws, but not usually its bounding box. Sustaining a pict takes a snapshot at the end of the pict's current duration and uses it for a new epoch inserted at the end of the current duration. For some animations, it is more natural to sustain by inserting a snapshot of the beginning of the last epoch in the current duration, just before that last epoch. The choice is deferred to the pict being sustained; the `animate` function exposes the choice through a `~sustain_edge` option that defaults to `TimeOrder.after`, but can be set to `TimeOrder.before`.

4.3 Fish Out of Water: Identity and Replacement

Because the underlying `Surface` already provides scaling and rotation, they are straightforward to add to a `Pict`, as we can just wrap a pict's drawing functions. Some other adaptations are beyond the horizon of the abstraction as presented, so far. An especially important adaptation, and the one remaining piece of our abstraction to explain, is replacement, giving DSL programmers the ability to change an existing pict by substituting another pict inside it with a different one.

There is a fundamental problem we must overcome to support replacement. So far, we've shown animated picts as a *shallowly embedded* domain-specific language. Each `Pict` operation generates a fresh `Pict` object, not storing an abstract syntax tree, but instead storing a black-box function that draws the `Pict`. This representation makes program transformations impossible, because we can't traverse the `Pict` to make changes after its construction. If we were to use a conventional *deep embedding*, we could effect replacement, as we could traverse the abstract syntax tree of the program that constructs a `Pict` to find the portion of the code that constructs some subpict and replace it with some new code.

A full-fledged deep embedding is a non-starter, however, as functions like `animate` and `animate_map` accept arbitrary Rhombus functions. In practice, users of pict need to add operators that use many features of the Rhombus language. Just from the earlier examples, `wiggle` involves significant arithmetic operations, and the `magic_move` operation uses finite maps and other sophisticated data structures to compute the diff between two picts. A fully deep embedding would require mirroring all of these expressive features in the pict DSL itself.

Still, the idea of a deeply embedded DSL is a helpful one, and our solution is to move in that direction by creating a hybrid between a shallow and a deep embedding. We'll still use first-class functions to represent some facets of a pict, but we'll also store a tree that tracks how it was constructed. That tree lets us adjust the construction of a `Pict` enough to support transformations such as replacement.

4.3.1 Pict Identity. Before we can talk about replacing one pict inside another, we need a notion of pict identity, so we can select the pict to replace. To enable finding picts within picts, we add a (time-invariant) `identity` field and a (time-varying) list of children picts. Each child `Pict` is paired with a `Transformation`, which is a 2-D affine transformation matrix that places, scales, and rotates a child pict within the enclosing pict.[4]

[4]We write an affine transformation in the usual order, `Transformation(`*sx*`, `*sxy*`, `*syx*`, `*sy*`, `*dx*`, `*dy*`)`, where *sx* is horizontal scaling, *syx* is a horizontal addition from the vertical component, *dx* is an additional horizontal transformation, and so on.

```
// fields and properties shown earlier elided here
class Pict(....,
           identity :: PictIdentity,
           children_at :: Time -> List.of(ChildPlacement))

class ChildPlacement(pict :: Pict, at :: Transformation)
```

Objects in Rhombus come with a primitive notion of identity similar to Scheme's notion of `eq?`, and we build on that to establish the identity of picts. Specifically, we create `PictIdentity` objects and store them on fields of `Pict`s, comparing these `PictIdentity` objects to determine the identity of the enclosing `Pict`. In most cases, a fresh `PictIdentity` is generated for every `Pict` construction because we are aiming for construction identity, and not structural identity. For example, if we have `def p1 = rectangle()`, `def p2 = rectangle()`, and `def p3 = beside(p1, p2)`, then `p1` should be found at a reliably distinct place from `p2`, even though `p1` and `p2` look identical.

A `beside` function for this expanded definition creates a fresh identity for a combined `Pict`, and it records the offset of each combined pict. This offset duplicates a transformation that is implicit in the composed `draw` function, but made explicit for the purpose of locating child `Pict`s.

```
fun beside(left :: Pict, right :: Pict) :: Pict:
  Pict(...., // fields shown earlier elided here
       PictIdentity(),
       [ChildPlacement(left, Transformation(1, 0, 0, 1, 0, 0)),
        ChildPlacement(right, Transformation(1, 0, 0, 1, left.width, 0))])
```

It's often useful to create a pict that has the same bounds and drawing as another, but with a fresh identity that hides all of its children. The `Pict.launder` method builds such a new one:

```
class Pict(....): // earlier methods and fields elided in this definition
  method Pict.launder() :: Pict:
    Pict(width_at, height_at, draw_at, duration, extent, rebuild, dependents,
         PictIdentity(), [])
```

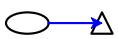
4.3.2 Finding Picts. Finding a pict within another is a matter of traversing `children` fields and composing `Transformations` on the way down. By default, an error is reported if a pict with a given identity can be reached multiple times with different transformations, such as when looking for `p1` in `beside(p1, p1)`.

Even before we consider replacing a pict inside of another, the ability to find a pict creates the possibility of useful drawing operations. The `connect` operation, for example, draws a line from one pict to another, optionally adding an arrow on one or both ends of the line. It accepts *from* and *to* descriptions of the locations of the arrows. These descriptions are represented as `Find` objects, which encapsulate `pict`-relative locations, e.g., the center of the left edge, the bottom right corner, or the center.

```
def p = beside(~sep: 20, ellipse_val, triangle_val)

> p
```

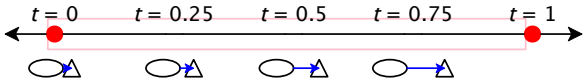
```
> connect(~on: p, ~style: ConnectStyle.arrow,
          Find.right(ellipse_val), Find.center(triangle_val),
          ~arrow_size: 5, ~line: "blue")
```



The location of a child pict is time-varying, because an animated pict might use the same children with different placements over time, and it may even combine different child picts at different times.

```
def anim_p = animate(fun (n :: Real.in(0, 1)):
                       beside(~sep: n * 20, ellipse_val, triangle_val))
def anim_conn = connect(~on: anim_p, ~style: ConnectStyle.arrow,
                        Find.right(ellipse_val), Find.center(triangle_val),
                        ~arrow_size: 5, ~line: "blue")
```

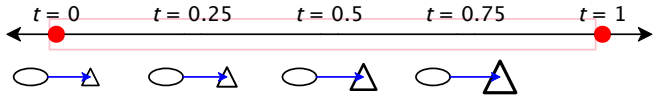
```
> explain_anim(anim_conn).scale(0.8)
```



Alternatively, the picts to find might be animated themselves, which may cause a pict's position to vary within a combination that automatically maps over animation snapshots.

```
def grow_triangle_val = animate(fun (n): triangle_val.scale(1 + n))
```

```
> explain_anim(
    connect(~on: beside(~sep: 20, ellipse_val, grow_triangle_val),
            Find.right(ellipse_val), Find.center(grow_triangle_val),
            ~style: ConnectStyle.arrow, ~arrow_size: 5, ~line: "blue")
  ).scale(0.8)
```



An interplay of snapshots and pict identity lurks beneath the surface of these examples. A snapshot of an animated pict is wrapped with one that has the same identity as the animation, which allows the animation to be found within the snapshot.

4.3.3 Rebuilding Picts. Keeping a list of children used to construct a pict enables finding a child, but it is not enough information to replace the child. The combined pict's drawing is represented by a Rhombus function, which cannot be inspected or modified to change a child pict's drawing. Even if drawing used an indirection to make individual steps replaceable, such as a dictionary that maps identities to drawing steps, replacing a pict with one that has a different bounding box or duration might change the transformations that other picts should use within the combined pict.

To support replacement, we extend `Pict` one last time with a `rebuild` function and a (time-invariant) list of *replaceable dependents* in `dependencies` as shown in figure 2. The `rebuild` function expects a list like `dependencies`, but with `Pict`s in the list potentially replaced, and it combines them into a result `Pict`. This protocol allows pict combinators to react to changes in their children. In essence, the `rebuild` function adds custom semantics to each node in the `Pict` tree, recording how the `Pict` was constructed and dictating all of its properties, including the `draw` function. Here's the extension for the `beside` function, making it sensitive to changes in its arguments by computing afresh the relative positions of the picts:

```
fun beside(left :: Pict, right :: Pict) :: Pict:
  Pict(...., // additional constructor arguments elided here
       fun ([left, right]): beside(left, right),
       [left, right])
```

```
class Pict(width_at :: Time -> Real,
           height_at :: Time -> Real,
           draw_at :: Time -> (Surface, Real, Real) -> Void,
           duration :: NonnegInt,
           extent :: Int -> Real,
           identity :: PictIdentity,
           children_at :: Time -> List.of(ChildPlacement),
           rebuild :: List.of(Pict) -> Pict,
           dependents :: List.of(Pict))
```

Fig. 2. Complete `Pict` datatype

The *replaceable dependents* of a pict in `dependents` are not necessarily the same as the *findable children* reported by `children_at`. The two lists are the same in the case of a combinator like `beside`, where child picts are always combined in the same way. In general, however, the child picts that can be found in an animated pict can vary over time, while the dependencies that can be replaced must be known up front. We discuss this difference more in section 4.3.5.

Rebuild operations are normally put in place by primitives such as `beside`. Accumulated `rebuild` operations in a `Pict` representation form something like a continuation: when a child is replaced, the computation continues with the replaced child. Although Rhombus provides first-class continuations, `rebuild` does not use them; it is difficult to reason about pervasive continuation capture and replay, and this more limited form of replay has served well enough in the `Pict` DSL.

When Rhombus `Pict` programmers need to add to the rebuild continuation, they use the `rebuildable` constructor, which gives them direct control over the `rebuild` operation. For example, this `hilite` function will not adapt the yellow rectangle correctly if `p` is replaced with a pict of different dimensions.

```
fun hilite(p :: Pict):
  overlay(rectangle(~width: p.width, ~height: p.height, ~fill: "yellow"), p)
```

The above version of `hilite` does not adapt to replacement, because it observes the width and height of the original `p` eagerly at the time of construction. In order to adapt to replacement, `hilite` needs to be replayed using the width and height of the replaced `p`. The solution is to use `rebuildable` to declare a dependency on `p` and put the use of `p.width` inside the `rebuild` function:

```
fun hilite(p :: Pict):
  rebuildable(
    [p], // dependencies
    fun ([p]): // rebuild function
      overlay(rectangle(~width: p.width, ~height: p.height, ~fill: "yellow"),
              p)
  )
```

The `rebuildable` function is needed whenever the programmer wants to create a new primitive that adapts to changes in children. These new primitives are powerful, allowing the programmer to extend the `Pict` language, but also more rare; usually, combinations of the existing provided primitives is sufficient. As a case in point, a better way to implement `hilite` in our `Pict` DSL is to use `rectangle` with an `~around` argument, which derives the rectangle's width and height from

the ~around argument and overlays it as in hilite. That way, the rectangle function takes care of the rebuilding.

Similarly, the animate function supports declaring a set of dependencies in order to support rebuilding an animation when one of its dependencies is replaced. For example, the wiggle function shown in section 3.3 can be modified to declare a dependency on the pict it wiggles.

```
fun wiggle(p :: Pict) :: Pict:
  animate(
    ~deps: [p],
    fun (n :: Real.in(0, 1), ~deps: [p]):
      def q = p.rotate(1/10 * math.sin(n * 2 * math.pi))
      overlay(p.ghost(), q.launder()).refocus(p),
  )
```

The rebuildable primitive can be implemented as follows:

```
fun rebuildable(deps :: List.of(Pict),
                rebuild :: List.of(Pict) -> Pict) :: Pict:
  def init_p = rebuild(deps)
  Pict(init_p.width, init_p.height, init_p.draw,
       init_p.duration, init_p.extent,
       PictIdentity(),
       [ChildPlacement(init_p, Transformation(1, 0, 0, 1, 0, 0))],
       rebuild,
       deps)
```

Like a shallow embedding would, the hybrid embedding approach of rebuildable preserves the full expressivity of Rhombus for animations like wiggle, new operators like hilite, and more general typesetting constructs like rhombusblock. At the same time, it introduces enough deep-embedding structure to let rebuild take on much of the burden of substituting picts. This hybrid approach comes at an implementation burden for some operations, specifically in manually specifying data dependencies via a deps argument, where failing to include relevant dependencies can prevent replace or rebuild from producing the intended results. Still we take advantage of the generality, for example to be able to substitute a wiggle result into a rhombusblock escape and still have the surrounding commas line up correctly.[5]

4.3.4 Replacing Picts. The Pict.replace method to replace from with to needs a helper function that traverses dependencies fields to replace from with to wherever it is found and to rebuild picts on the path to each use of from. The implementation in figure 3 shows several key points:

- (line 4) A cache map is used to avoid building a pict multiple times.
- (line 11-12) A pict is not rebuilt if it does not transitively depend on from.
- (line 18-19) A rebuilt pict is wrapped to that rebuilding again uses the same rebuild function as the original, skipping over a previous rebuild result.
- (line 20) A rebuilt pict is wrapped to record the original pict's identity, so that searching for the original pict finds its replacement.

Recall that the small_step function from section 3.5 needs Pict.find_rebuilt to correctly wiggle the next subexpression to evaluate: the subexpression to wiggle may have been rebuilt itself, replacing more nested subexpression picts with their values as picts. That latest version of the

[5]The rhombusblock form needs to do more than horizontal combination with baseline alignment, because a comma may need to appear after a substituted expression's final line, which may be shorter than the expression's preceding lines.

```
1  fun replace_help(in_p :: Pict,
2                   from :: Pict, to :: Pict,
3                   cache :: Map.of(Pict,Pict)) :: [Pict, Map.of(Pict,Pict)]:
4    cond
5    | in_p == from: [to, cache]
6    | in_p in cache: [cache[in_p], cache]
7    | def [new_deps, new_cache]:
8        // fold `replace_help` over `in_p.dependencies`:
9        for fold ([new_deps, cache] = [[], cache]):
10         each dep in in_p.dependencies
11         def [new_dep, new_cache] = replace_help(dep, from, to, cache)
12         [new_deps.add(new_dep), new_cache]
13     if new_deps == deps
14     | [in_p, cache.add(in_p, in_p)]
15     | def rebuilt_p = in_p.rebuild(new_deps)
16       def new_p:
17         Pict(rebuilt_p.width, rebuilt_p.height, rebuilt_p.draw,
18              rebuilt_p.duration, rebuilt_p.extent,
19              in_p.rebuild,  // keep rebuild of `in_p`
20              new_deps, in_p.identity, // wrap with `in_p` identity
21              [ChildPlacement(new_p, Transformation(1, 0, 0, 1, 0, 0))])
22       [new_p, cache.add(in_p, new_p)]
```

Fig. 3. Helper function for `Pict.replace`

subexpression is the right one to pass to `wiggle`. The last bullet above (line 20 in figure 3) makes the `Pict.find_rebuilt` method possible, where it can find a replacement pict through a simple tree walk that compares pict identities:

```
method Pict.find_rebuilt(p :: Pict) :: maybe(Pict):
  if p.identity == identity
  | this
  | for any (c :: Pict in children):
      c.find_rebuilt(p)
```

The Rhombus `Pict` library further generalizes `Pict.replace` by a `Pict.rebuild` method (not to be confused with the `rebuild` internal field) to accept both pre-replacement and post-replacement updates on a dependency, and a pict created by `rebuildable` can have an arbitrary key–value map among its dependencies where values can be updated along the same lines as replacing a dependent pict. Those generalizations build on the ideas we have presented, and the `Pict.replace` operation remains by far the most common use of `Pict.rebuild`.

4.3.5 Findable Children versus Replaceable Dependencies. As noted in section 4.3.3, a pict's findable children can be different from its replaceable dependencies. For an animated pict or the result of `rebuildable`, the findable children are determined by the result of the `sample` function provided to `animate` or the `rebuild` function provided to `replaceable`. That choice is possible because finding is always relative to a point in time, and so the `sample` and `rebuild` functions can be sampled as needed. Replacement may change the duration of a pict, however, so sampling is not an option; replaceable dependencies must be declared up front.

This distinction sets up an ambiguity in the case of the `Pict.snapshot` function, because there are two possible interpretations of replacing in a snapshot. Replacement could be performed on the pict before a snapshot is taken, or it could be applied to the snapshot itself, ignoring the composition operators that produced the animated Pict whose snapshot is taken. The latter is a form of *delimiting* the replace operation at a snapshot boundary.

Both options are useful, and `Pict.snapshot` can be configured for one or the other through a `~rebuild_prompt` optional argument. The default is `#true`, the delimiting mode, because that is the intent in most cases. Delimiting is perhaps more intuitive behavior, along the same lines as replacing Alice with Bob in a photograph—by which someone would surely mean replacing a still of Alice with a still of Bob, not swapping Alice and Bob at birth and hoping that Bob's choices in life would lead him to pose in the same place and the same time as Alice did.

5 Related Work

In this section, we situate our work in three related strands. We first compare it to picture and animation systems, then to literature on shallow and deep embeddings, and finally to hybrid embedding patterns in practice.

5.1 Picture and Animation Systems

Most 2-D graphical environments offer a native drawing interface based on PostScript (Adobe Press 1985). That's a much higher level of abstraction than the array of pixels provided by a physical device, but still low-level in the sense that commands perform imperative actions over numerical locations. Although a composite drawing can be represented as a function that encapsulates a sequence of commands, and although those functions can be composed by calling them in sequence, plain functions offer limited composition opportunities. For example, there's no general way to stack two drawings vertically, because a drawing function by itself does not indicate how much vertical space it needs.

Existing drawing libraries provide a more functional interface with richer set of composition operators (Kernighan 1991; Kamin and Hyatt 1997; Henderson 1982; Finne and Peyton Jones 1995). In those libraries, a picture is more than just a sequence of drawing commands. A picture includes width and height information, at a minimum, to enable vertical and horizontal stacking, and it may have additional properties depending on the specific graphical domain. In the case of Racket's Slideshow library (Findler and Flatt 2004), each picture also knows the (x,y) location of its component pictures to enable post-hoc placement of more pictures relative to existing ones. Our library builds directly on those designs. A more recent example in the same compositional tradition is PyTamaro (Chiodini et al. 2023), a Python library aimed at teaching beginners, which similarly treats pictures as values and combines them through high-level operations rather than coordinate-based drawing commands.

Abstractions for animated pictures go back about as far as abstractions for static pictures (Pfister 1976; T. J. O'Donnell and Olson 1981; Hudak 1998; Zongker and Salesin 2003). A functional language for static pictures can be used for animations by parameterizing a picture's creation with respect to time, but that leads back to the limitations of a function as an abstraction. As observed by Matlage and Gill (2010), an abstraction for composable animations needs time bounds that are analogous to the width and height of its graphical component. Our library adapts that to the particular notion of time needed for slide presentations, which involves both presenter-triggered slide transitions and real-time animations within a slide.

Draft-standard CSS (WC3 Group 2024) supports animation with similar goals to slide-presentation animations, where events such as opening the page or clicking an element are similar to advancing a slide. Animation is expressed in terms of transitions from one key-frame style to another, and the

implicit animation function is a time-varying interpolation between the two. Visualization tools such as D3 (Observable 2011) and Vega Lite (Satyanarayan et al. 2016; Zong et al. 2022) similarly incorporate support for animated and interactive charts through declarative specifications. In those visualization tools, the time-varying aspect of an animation often reflects a dimension of the data being visualized, and so it is natural to allow users to directly control the time parameter as a way of scanning through data. These languages are standalone DSLs, so the `Pict` library is different in its emphasis on making use of a general-purpose host language as well as the kinds of animations that it targets.

5.2 Shallow and Deep Embeddings in the Literature

Building on the shallow/deep characterizations introduced in section 2.1, the literature offers a variety of design patterns for combining the two in a single embedded DSL. Boulton et al. (1992)'s setting—a higher-order logic—shows that deep embeddings allow the DSL author to reason over classes of programs, but require significant extra work in order to set up the semantics of the DSL explicitly. In our context, we do not need to reason about `Pict` compositions via Rhombus functions (e.g., to optimize them), and we make free use of the host language, so a strict reading of this definition would treat our embedding as shallow.

Axelsson et al. (2010), in their discussion of Feldspar, discuss a combination of shallow and deep embeddings, too. Their combination privileges the deep embedding, and the ideas of shallow embeddings are used to extend the DSL. That is, they write host language functions that operate on the data structure in the deep embedding. Programmers from the Scheme and Racket community would more easily recognize this combination of deep and shallow embeddings as a deep embedding plus macro support for the embedded DSL, as functions in the host language operate one phase before the embedded language in the deep setting and benefit from support for managing variable references.

Claessen (2001) shows how to simultaneously achieve a deep and a shallow embedding for the same language using overloading via Haskell's type classes. This technique enables the DSL programmer to delay the choice between the two embedding techniques and to reap the benefits of simple, host-language supported evaluation (reusing debugging support, for example) via the shallow embedding and also sophisticated analysis and property verification via the deep embedding. This analysis still treats deep and shallow as a choice, where the innovation in this work is to simultaneously support both choices, as opposed to viewing deep and shallow as extreme points on a spectrum.

Jovanovic et al. (2014) and Rompf et al. (2013) also discuss the dichotomy between deep and shallow, and they work to smooth it over, aiming to give DSL programmers the fluency of a shallow embedding and the performance of a deep embedding. This notion of *shallow* and *deep* is consistent with the original meaning, and it hints at the design-level interpretation that we have applied to the terms, but their primary focus is performance via parallelism and that focus drives their design. Additionally, this treatment of the terms maintains the perspective that the choice is a binary one.

As section 2.1 describes, Gibbons and Wu (2014) characterize the gap between shallow and deep as filled by fold operators, enabling a systematic conversion between the two styles. Matsuda et al. (2023) demonstrate an especially sophisticated take, showing how to support DSLs with urbane operations (like running programs backwards) by converting back and forth between deep (finally tagless) and shallow (HOAS) representations in a DSL.

Elliott et al. (2000)'s DSL for image manipulation, Pan, has echoes of our perspective on DSLs. Although the paper does not contain the either of the words "deep" or "shallow," they run into a quandary that parallels the experience with our design, but from the opposite direction. Unlike ours, Pan's implementation starts from a purely deep DSL, as the ability to manipulate image programs

DSL / host	Shallow	Deep
`Pict` / Rhombus	`wiggle`, `animate` combinators	pict AST via `rebuildable`
PyTorch / Python	eager Python tensor code	traced graphs via `torch.compile`
Spark / Scala or Python	closures passed to `map`, `filter`	transformation DAG over RDDs
React / JavaScript	JS expressions inside components	reified component tree

Fig. 4. Shallow and deep elements of embedded DSLs across the spectrum

before executing them takes programs from impossibly inefficient to performant. In that context, they observe that adding functions to the DSL is complex and misses the opportunity to reuse the implementation of functions in the host language, effectively desiring what would be easy to add with a shallow embedding. They cleverly extend their deep DSL with a construct that enables them to inspect *host* functions on DSL programs and then come up with clever solutions to ensuing technical problems, enabling reuse in the DSL while still getting good performance.

5.3 Hybrid Shallow and Deep DSLs in Practice

Hybrid designs that combine elements of shallow and deep embeddings are common in practice. Shallow embeddings most readily accommodate new operations, while deep embeddings more readily accommodate new interpretations. As a DSL implementation grows, however, it may need both kinds of extensibility, and users of the DSL may need to introduce both new operations and new interpretations. In Figure 4, we categorize the embedded DSLs discussed in this section by their shallow and deep elements, alongside the corresponding choices in our own `Pict` design. The rest of the section walks through each row in turn.

In the case of picts, we found many useful interpretations on slideshows (such as replacement or magic move) and many useful operations (such as new shapes or new ways to compose picts). A shallow DSL makes adding new shapes and pict compositions easy. A deep DSL enables the replacement operation, which inspects the way a pict is constructed in order to rebuild the pict after the replacement. Although this might seem to leave us somewhere between the devil and the deep blue sea, there is a middle ground. We annotate the AST with host-level functions and an explicit list of dependencies, allowing new operations to reveal their structure and contribute to the rebuilding process via the `rebuildable` constructor.

Stepping back from `Pict` specifically, we see the deep/shallow dichotomy not as a binary choice, but as a spectrum where competing design considerations push the DSL designer toward or away from both deep and shallow designs. Many successful systems land somewhere in the middle.

In the context of machine learning, deep embeddings are crucial for enabling global optimization of computation graphs, but they restrict direct use of host-language constructs. On one hand, shallow embeddings support the full host language out-of-the-box while enabling cheap observation and debugging of intermediate values. On the other hand, deep embeddings enable new interpretations, most importantly optimization of computation graphs. Early versions of TensorFlow (the 1.x series) enforced a graph-based execution model, requiring all control flow and iteration to be expressed using dedicated DSL constructs such as `tf.cond` and `tf.while_loop` (Abadi et al. 2016). In contrast, PyTorch adopted an eager execution model that allowed users to write arbitrary Python code while constructing computation graphs dynamically (Paszke et al. 2019). Later versions of both TensorFlow and PyTorch introduced mechanisms for switching between eager execution and graph-based execution, including tracing and compiling subsets of Python code into computation graphs to enable optimization and deployment (Agrawal et al. 2019; Moldovan et al. 2019). PyTorch's `torch.compile` function (Ansel et al. 2024) stores fragments of Python code as

optimized computation graphs (resembling a deep embedding) but executes normal Python code during graph breaks (resembling a shallow embedding). Similarly, Numba (Lam et al. 2015) lets a programmer mark individual functions with the `@jit` decorator, and traces and compiles those bodies to optimized machine code while ordinary Python continues to run around them.

In the domain of data processing, embedded DSLs choose solutions between shallow and deep with far-reaching consequences for expressivity and performance. Systems such as MapReduce choose a shallow approach, fixing the computation graph up front, but allowing arbitrary host-language functions to process data (Dean and Ghemawat 2004). MapReduce sacrifices the potential for multiple interpretations (such as global query optimization) in favor of local computation flexibility and a fixed dataflow structure. Apache Spark is more expressive than MapReduce through a hybrid embedding allowing users to write host-language functions over a DAG-based abstraction (Zaharia et al. 2012). Apache Spark's hybrid approach enables optimizations over the DAG while preserving flexibility of user-defined functions locally. On the deep end, DuckDB significantly enriches SQL with features such as recursive common table expressions, complex and nested data types, window functions, and vectorized execution (DuckDB Contributors 2026; Raasveldt and Mühleisen 2019). DuckDB performs whole-query optimization and compilation by analyzing the complete SQL query structure, but no host-language code is embedded within SQL queries.

JavaScript libraries and frameworks for interactive webpages meet expressivity challenges with hybrid embedding techniques as well. Notably, the React framework provides a traversable abstraction called a component, which has sub-components (Meta Platforms, Inc. 2013). Each component is parameterized by JavaScript closures, making React a hybrid embedding. React is extremely popular due to its convenient abstractions (components), expressivity (arbitrary JavaScript code in closures), and performance (efficient updates to the DOM).

6 Conclusion

We have implemented a new pict abstraction that supports animation and that also includes a unique notion of pict identity and replacement. The implementation is available as the `rhombus-pict` Racket/Rhombus package. Our earliest designs (Findler and Flatt 2004) aimed for minimalism, based on the idea that the host language's rich abstraction facilities would give us all the flexibility that we'd ever need. A reassessment lead us away from an *either-or* allocation of abstraction, however, and toward a *both-and* use of host language constructs and an enriched picture abstraction. Our resulting design could be viewed as an intermediate point on the spectrum between a *deep* and *shallow* DSL embedding. We offer our work both as a recommended abstraction for slide animation and as an additional point of view on the distinction between deep and shallow DSLs.

We return to the harbor with an interesting open problem. As each step in `Pict`'s design increases the expressiveness of the DSL, perhaps a variation of Felleisen (1991)'s notion of expressiveness applies to DSLs. Future work could quantify the expressiveness of embedded DSLs and how it relates to deep vs. shallow embeddings. We expect that a technically precise characterization would be useful in the design of future embedded DSLs and understanding existing ones.

Data-Availability Statement

An implementation and examples are available in the paper's artifact (Flatt et al. 2026).

Acknowledgments

Thanks to the anonymous ICFP reviewers for their help improving the exposition and for the contribution of a few puns. This work was supported in part by NSF grants CCF-2421308 and CCF-2426443 and by NSF Graduate Research Fellowship DGE-2140004. The dragon curve example was inspired by a talk by Shin-Cheng Mu.

A Drawing the Sea Dragon

Continuing from section 2.2, the curve returned by `sea_dragon` can be drawn in imperative "turtle graphics" style by treating each index in the result list as an instruction: move forward and draw by one drawing unit, then turn right or left depending on whether the command is `0` or `1`. To make that imperative process functional, we can represent the turtle state as a list of four values: the pict accumulated so far, the turtle's x-position, the turtle's y-position, and the turtle's θ heading.

Pens do not work well under water, so let's have a sea turtle drop a fish for each segment. We need a function to get a fish that is rotated around its nose opposite to the turtle's heading, θ:

```
fun rotated_fish(θ :: Real) :: Pict:
  def nose = blank()
  beside(nose, fish)
    .refocus(nose)
    .rotate(-θ)
```

This function uses a trick to ensure the fish rotates around its nose. If we used `rotate` directly on the fish it would rotate around the center so, instead, we put a zero-sized pict directly on the spot we wish to rotate around and `refocus` on that pict. This pattern of creating a zero-sized pict that, nevertheless, draws some larger image is also handy for composing pictures, as it does not perturb the existing drawing elements while still adding something to the canvas.

The drawing of the entire picture uses the same idea, as the accumulated pict, `p`, keeps its bounding box around the origin. Each drawing step `overlay`s a fish at the turtle's x- and y-position and then refocusing the result brings the accumulated drawing's bounding box back to the origin. Ending with `refocus_around_ink` makes the final bounding box surround all the fish.

```
fun dragon_pict(n :: Int) :: Pict:
  def [curve, _, _, _]:
    for fold([p, x, y, θ] = [blank(), 0, 0, 0]):
      each i in sea_dragon(n)
      def nx = x + math.cos(θ)*fish.width
      def ny = y + math.sin(θ)*fish.width
      [overlay(p, ~dx: x, ~dy: y, rotated_fish(θ))
         .refocus(p),
       nx, ny,
       θ - (i*2-1)*math.pi/2]
  curve.refocus_around_ink()
```

Here's the output of `dragon_pict(11).scale(1/20)`:

References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. In *Proc. Operating Systems Design and Implementation*, 2016.

Adobe Press. PostScript Language Reference Manual. 1985.

Akshay Agrawal, Akshay Modi, Alexandre Passos, Allen Lavoie, Ashish Agarwal, Asim Shankar, Igor Ganichev, Josh Levenberg, Mingsheng Hong, Rajat Monga, and Shanqing Cai. TensorFlow Eager: A Multi-Stage, Python-Embedded DSL for Machine Learning. In *Proc. Machine Learning and Systems*, 2019. https://proceedings.mlsys.org/paper_files/paper/2019/hash/b3cd73d353d39e5cf6f6e9ff8d14c87f-Abstract.html

Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proc. Architectural Support for Programming Languages and Operating Systems*, 2024. doi:10.1145/3620665.3640366

Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. The Design and Implementation of Feldspar an Embedded Language for Digital Signal Processing. In *Proc. International Conference on Implementation and Application of Functional Languages*, 2010. doi:10.1007/978-3-642-24276-2_8

Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with Embedding Hardware Description Languages in HOL. In *Proc. International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, 1992.

Luca Chiodini, Juha Sorva, and Matthias Hauswirth. Teaching Programming with Graphics: Pitfalls and a Solution. In *Proc. SPLASH-E*, 2023. doi:10.1145/3622780.3623644

Koen Claessen. Embedded Languages for Verifying Hardware. PhD dissertation, Chalmers University, 2001.

William R. Cook. Object-Oriented Programming Versus Abstract Data Types. In *Proc. REX School/Workshop on Foundations of Object-Oriented Languages*, 1990.

Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. Operating Systems Design and Implementation*, 2004. https://www.usenix.org/conference/osdi-04/mapreduce-simplified-data-processing-large-clusters

DuckDB Contributors. Window Functions. 2026. https://duckdb.org/docs/sql/window_functions

Conal Elliott, Sigbjorn Finne, and Oege de Moor. Compiling Embedded Languages. In *Proc. International Workshop on Semantics, Applications, and Implementation of Program Generation*, 2000.

Matthias Felleisen. On the Expressive Power of Programming Languages. *Science of Computer Programming* 17(1-3), 1991. doi:10.1016/0167-6423(91)90036-W

Robert Bruce Findler and Matthew Flatt. Slideshow: Functional Presentations. *Journal of Functional Programming* 16(4-5), 2004. doi:10.1017/S0956796806006010

Sigbjorn Finne and Simon Peyton Jones. Pictures: A Simple Structured Graphics Model. In *Proc. Glasgow Workshop on Functional Programming*, 1995. doi:10.14236/ewic/FP1995.6

Matthew Flatt, Taylor Allred, Nia Angle, Stephen De Gabrielle, Robert Bruce Findler, Jack Firth, Kiran Gopinathan, Ben Greenman, Siddhartha Kasivajhula, Alex Knauth, Jay McCarthy, Sam Phillips, Sorawee Porncharoenwase, Jens Axel Søgaard, and Sam Tobin-Hochstadt. Rhombus: A New Spin on Macros Without All the Parentheses. In *Proc. Object-Oriented Programming, Systems, Languages and Applications*, 2023. doi:10.1145/3622818

Matthew Flatt and Robert Bruce Findler. Modular Object-Oriented Programming with Units and Mixins. In *Proc. International Conference on Functional Programming*, 1998. doi:10.1145/291251.289432

Oliver Flatt, Robert Bruce Findler, and Matthew Flatt. Animated Pictures for Slide Presentations (Functional Pearl) Artifact. 2026. doi:10.5281/zenodo.20329086

Jeremy Gibbons and Nicolas Wu. Folding Domain-Specific Languages: Deep and Shallow Embeddings (Functional Pearl). In *Proc. International Conference on Functional Programming*, 2014. doi:10.1145/2692915.2628138

Peter Henderson. Functional Geometry. In *Proc. Lisp and Functional Programming*, 1982. doi:10.1145/800068.802148

Paul Hudak. Modular Domain Specific Languages and Tools. In *Proc. International Conference on Software Reuse*, 1998.

Vojin Jovanovic, Amir Shaikhha, Sandro Stucki, Vladimir Nikolaev, Christoph Koch, and Martin Odersky. Yin-Yang: Concealing the Deep Embedding of DSLs. In *Proc. Generative Programming: Concepts and Experiences*, 2014. doi:10.1145/2658761.2658771

Samuel N. Kamin and David Hyatt. A Special-Purpose Language for Picture-Drawing. In *Proc. USENIX Conference on Domain-Specific Languages*, 1997.

Brian W. Kernighan. PIC — a Graphics Language for Typesetting, User Manual. AT&T Bell Laboratories, CSTR-116, 1991.

Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. Synthesizing Object-Oriented and Functional Design to Promote Re-Use. In *Proc. European Conference on Object-Oriented Programming*, 1998.

Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A LLVM-based Python JIT Compiler. In *Proc. Workshop on the LLVM Compiler Infrastructure in HPC*, 2015. doi:10.1145/2833157.2833162

Kevin Matlage and Andy Gill. Every Animation Should Have a Beginning, a Middle, and an End. In *Proc. Trends in Functional Programming*, 2010. doi:10.1007/978-3-642-22941-1_10

Kazutaka Matsuda, Samantha Frohlich, Meng Wang, and Nicolas Wu. Embedding by Unembedding. In *Proc. International Conference on Functional Programming*, 2023. doi:10.1145/3607830

Meta Platforms, Inc. React: A JavaScript Library for Building User Interfaces. 2013. https://react.dev/

Dan Moldovan, James Decker, Fei Wang, Andrew Johnson, Brian Lee, Zachary Nado, D. Sculley, Tiark Rompf, and Alexander B. Wiltschko. AG: Imperative-style Coding with Graph-based Performance. In *Proc. Machine Learning and Systems*, 2019. https://proceedings.mlsys.org/paper_files/paper/2019/hash/e31cef6b735cf838db79202dbee7b093-Abstract.html

Observable. D3. 2011. https://d3js.org/

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proc. Neural Information Processing Systems*, 2019.

Gregory F. Pfister. A High Level Language Extension for Creating and Controlling Dynamic Pictures. In *Proc. ACM SIGPLAN/SIGGRAPH Symposium on Graphical Languages*, 1976. doi:10.1145/957199.804725

Mark Raasveldt and Hannes Mühleisen. DuckDB: an Embeddable Analytical Database. In *Proc. SIGMOD Conference*, 2019. doi:10.1145/3299869.3320212

John C. Reynolds. User-Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction. In *Proc. New Directions in Algorithmic Languages*, 1975.

Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. Scala-Virtualized: Linguistic Reuse for Deep Embeddings. *Higher-Order and Symbolic Computation* 25, 2013. doi:10.1007/s10990-013-9096-9

Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. Vega-Lite: A Grammar of Interactive Graphics. In *Proc. IEEE Transactions on Visualization and Computer Graphics*, 2016. doi:10.1109/TVCG.2016.2599030

T. J. O'Donnell and Arthur J. Olson. GRAMPS - A Graphics Language Interpreter for Real-Time, Interactive, Three-Dimensional Picture Editing and Animation. *Computer Graphics* (15), 1981. doi:10.1145/965161.806799

WC3 Group. CSS Snapshot 2024. 2024. https://www.w3.org/TR/2025/NOTE-css-2024-20250225/

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proc. Networked Systems Design and Implementation*, 2012.

Jonathan Zong, Josh Pollock, Dylan Wootton, and Arvind Satyanarayan. Animated Vega-Lite: Unifying Animation with a Grammar of Interactive Graphics. In *Proc. IEEE Transactions on Visualization and Computer Graphics*, 2022. doi:10.1109/TVCG.2022.3209369

Douglas E. Zongker and David H. Salesin. On Creating Animated Presentations. In *Proc. Eurographics/SIGGRAPH Symposium on Computer Animation*, 2003. doi:10.2312/SCA03/298-308

Received 2026-02-19; accepted 2026-05-13