PaSe: An Extensible and Inspectable DSL for Micro-Animations

Ruben P. Pieters ✉[0000−0003−0537−9403] and Tom Schrijvers[0000−0001−8771−5559]

KU Leuven, 3001 Leuven, Belgium
{ruben.pieters, tom.schrijvers}@cs.kuleuven.be

First author is a *research student*. Article category: *Project Article*.

Abstract. This paper presents PaSe, an extensible and inspectable DSL embedded in Haskell for expressing micro-animations. PaSe builds animations in compositional fashion, using parallel and sequential animations as basic building blocks. This differs from typical animation libraries which mostly focus on sequential composition and utilize callbacks and implicit effects for their expressivity. To provide similar flexibility to other animation libraries, PaSe features extensibility of operations and inspectability of animations. We present the features of PaSe with a to-do list application, discuss the PaSe implementation, and argue that the callback style of extensibility is detrimental for correctly integrating inspectability. To illustrate this, we contrast with the GreenSock Animation Platform, a professional-grade and widely used JavaScript animation library.

1 Introduction

Because of their ability to structure effectful code in a pure functional codebase, monads quickly became ubiquitous in functional programming [20]. They have since seen wide use in Haskell Domain Specific Languages (DSLs). However, the choice for a monadic DSL implies certain trade-offs. The obvious advantage of monadic DSLs is their expressivity, but there are also drawbacks. The main loss is that of *inspectability*, as monadic computations can only be inspected up to the next action. Techniques such as applicative functors [16], arrows [9], or selective applicative functors [18] choose the other side of the trade-off: they increase the inspection capabilities by reducing the expressivity compared to monads.

This paper develops a DSL embedded in Haskell for defining micro-animations, called PaSe[1]. PaSe employs a technique which alleviates some aspects of the trade-off between expressivity and inspectability. The expressivity of control flow is restricted by means of type classes, inspired by the MTL style originally introduced by Liang *et al.* [14]. The MTL style is an open encoding which allows extensions to the syntax of the DSL. Instantiating the abstract animation definitions with, for example, the `Const` functor provides inspectability. Expressivity

[1] Pronounced *pace* (peɪs), the name is derived from ***Pa****rallel* and ***Se****quential*.

can be increased, while preserving inspectability, by adding new control flow constructs to the DSL and providing a corresponding instance for inspection.

Micro-animations are short animations displayed when users interact with an application, for example an animated transition between two screens. When used appropriately, they aid the user in understanding evolving states of the application [1,7,8]. Examples can be found in almost every software application: window managers animate window minimization, menus in mobile applications pop in gradually, browsers highlight newly selected tabs with an animation, etc.

PaSe provides the features expected of animation libraries by building them with recent ideas from functional programming. Our contributions are as follows:

- We develop PaSe, which supports arbitrary composition of animations and inspectability. Animation libraries, such as the GreenSock Animation Platform (GSAP)[2], typically use callbacks as a means of extensibility/expressivity; this is detrimental to inspectability. We show an example resulting in unexpected behaviour and how PaSe correctly handles it.
- PaSe is an *extensible* DSL: the syntax can be extended with new operations. The animations use case is novel for approaches to extensibility.
- PaSe supports *inspectability*: extracting information from computations before running it. Inspectability is present in specific computation classes, such as free applicatives [2]. But, it is novel to combine it with extensibility.
- PaSe supports arbitrary nesting of parallel and sequential animations which correctly interacts with inspectability. Such parallel components exist already, see for example Ren'Py[3], React Native Animations[4] or Qt Animations[5]. Yet, general-purpose animation libraries lack them. Also, we correctly support the interaction with inspectability.
- We implemented various examples[6]: a to-do list application, a communication story example, a game-like demo application and a Pac-Man game. We combined PaSe with both `gloss`[7] and the Haskell SDL bindings[8] as graphics backend. This paper uses the to-do list as motivating application and compares the development of the Pac-Man application, developed in both Haskell with PaSe and in TypeScript with GSAP.

2 Motivation

We present a to-do list application to showcase the functionality of PaSe.

[2] https://greensock.com
[3] https://www.renpy.org/doc/html/atl.html#parallel-statement
[4] https://facebook.github.io/react-native/docs/animated#parallel
[5] https://doc.qt.io/qt-5/animation.html
[6] https://github.com/rubenpieters/PaSe-hs/tree/master/PaSe-examples
[7] https://hackage.haskell.org/package/gloss
[8] https://hackage.haskell.org/package/sdl2

2.1 Running Example

Our application has two screens: a main screen and a menu screen. The main screen contains a navigation bar and three items. An overview of the application is given in Figure 1. These screenshots are captured from the application built by combining PaSe with `gloss` as graphics backend.

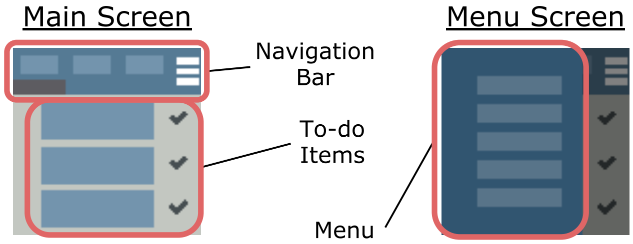


Fig. 1: Overview of the to-do list application.

In this application, various user actions are accompanied with an animation. We list these actions below. Some animations are shown in Figure 2.

- The user marks items as *(not) done* by clicking them. The checkmark icon changes shape and color to display its status change.
- The user filters items by their status with the navigation bar buttons. The leftmost shows all items, the middle shows all completed items, and the rightmost shows all unfinished items. The navigation bar underline and to-do items itself change shape to indicate the new selection.
- The menu screen shows/hides itself after clicking the menu icon (≡). The menu expands inwards from the left, to indicate the application state changes.

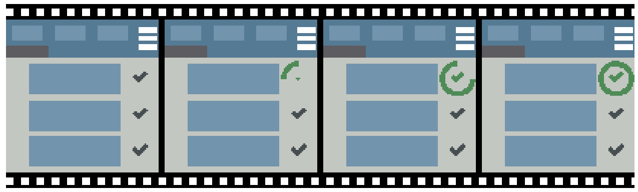
2.2 Composing Animations

Animations are built in a compositional fashion. When creating an animation, we decompose it into smaller elements. For example, the `menuIntro` animation both introduces the menu screen and fades out the background. Thus, it is composed of two basic animations `menuSlideIn` and `appFadeOut` in parallel. The next sections explain how to construct such basic and composed animations.

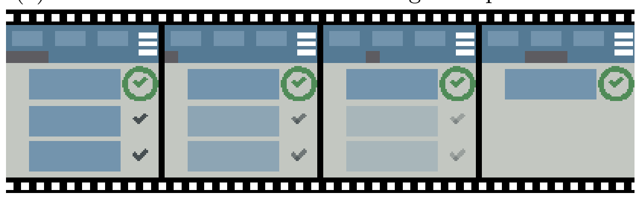
Basic Animations Basic animations change the property of an element over a period of time. The `linearTo` function has three inputs: a lens targeting the property, the duration, and the target value for this property. This results in a linear change from the current value to its target, hence the name. The duration is specified with `For` while the target value is specified with `To`.

To animate the navigation bar underline, we reduce the width of the leftmost underline for 0.25 seconds and increase the width of the middle underline for 0.25 seconds. These animations are expressed in respectively `line1Out` and `line2In` below, and visualized in Figure 3.

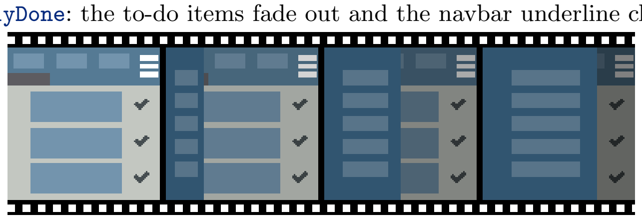
```
line1Out = linearTo (navbar . underline1 . width) (For 0.25) (To 0)
line2In = linearTo (navbar . underline2 . width) (For 0.25) (To 28)
```



(a) `markAsDone`: the checkmark changes shape and color.



(b) `onlyDone`: the to-do items fade out and the navbar underline changes.



(c) `menuIntro`: the menu appears while the background fades out.

Fig. 2: Micro-Animations in the to-do list application.

Note on Lenses We use lens notation `x . y . z` to target `z` inside a nested structure `{ x: { y: { z: T } } }`. This type of lenses was conceived by van Laarhoven [13], and later packaged into various Haskell libraries, such as `lens`[9].

The `menuSlideIn` and `appFadeOut` animations are other examples. For the former, we increase the width of the menu over a duration of 0.5 seconds, and for the latter we increase the opacity of the obscuring box, determined by `alpha`, over a duration of 0.5 seconds. These animations are visualized in Figure 3.

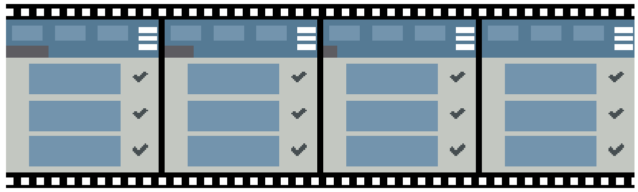
```
menuSlideIn = linearTo (menu . width) (For 0.5) (To 75)
appFadeOut = linearTo (obscuringBox . alpha) (For 0.5) (To 0.65)
```

Composed Animations A composed animation combines several other animations into a new one. We can do this either in *sequence* or in *parallel*.

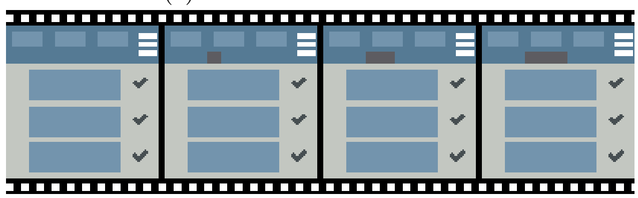
We create `selectBtn2` by combining `line1Out` and `line2In` with `sequential`. This constructs a new animation which first plays `line1Out`, and once it is finished plays `line2In`.

```
selectBtn2Anim = line1Out `sequential` line2In
```

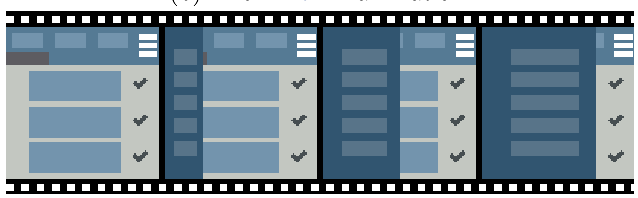
[9] https://hackage.haskell.org/package/lens



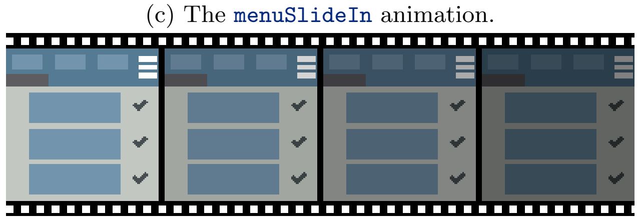
(a) The `line1Out` animation.



(b) The `line2In` animation.



(c) The `menuSlideIn` animation.



(d) The `appFadeOut` animation.

Fig. 3: Basic `linearTo` animations.

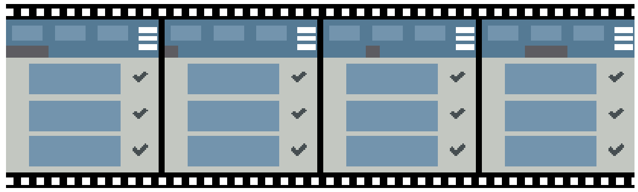
To obtain `menuIntro`, we combine both `menuSlideIn` and `appFadeOut` with `parallel`. This constructs a new animation which plays both `menuSlideIn` and `appFadeOut` at the same time.

```
menuIntro = menuSlideIn `parallel` appFadeOut
```

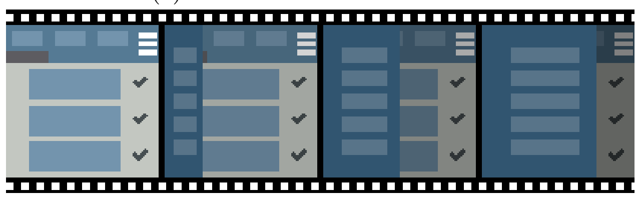
Both of these animations are visualized in Figure 4.

3 Extensibility, Inspectability and Expressiveness

The features in Section 2 form the basis of PaSe. To provide support for additional features present in other animation libraries, we design PaSe to be extensible and inspectable. This means that PaSe can be extended with new operations and information can be derived from inspecting specified animations. To support arbitrary expressiveness in combination with those features, we also emphasize the possibility to extend PaSe with new combinators.



(a) The `selectBtn2` animation.



(b) The `menuIntro` animation.

Fig. 4: All of the defined composed animations.

3.1 Extensibility

The `linearTo` operation and the `sequential` and `parallel` combinators form the basis for expressing a variety of animations. However, there are situations which require other primitives to express desired animations. For example, GSAP provides a primitive to morph one shape into another.

An example in the to-do list app is `checkIcon`, part of `markAsDone`, where we want to set the color of the checkmark to a new value. We define a custom `set` operation and embed it inside a PaSe animation. In this animation we use Haskell's `do`-notation to specify sequential animations.

```
checkIcon = do ...; set (checkmark . color) green; ...
```

3.2 Inspectability

PaSe is inspectable, meaning that we can derive properties of expressed computations by *inspecting them rather than running them*. For example, we want to know the duration of `menuIntro` without actually running it and keeping track of the time. The `duration` function calculates the duration by inspecting the animation. Passing it `menuIntro` gives a duration of `0.5` seconds, which is indeed the duration of two `0.5` second animations in parallel.

```
menuIntroDuration = duration menuIntro -- = 0.5
```

Of course, it is not possible to inspect every animation. In the following situation we have a custom operation `get`, the dual of `set` in the previous section, returning a `Float`. If the result of this value is used as the duration parameter, then we cannot know upfront how long this animation will last. Requesting to calculate the duration then results in a type error.

```
complicatedAnim = do v <- get; linearTo lens (For v) (To 10)
complicatedAnimDuration = duration complicatedAnim -- type error
```

Calculating a duration is a stepping stone towards other interesting features. One such example is sequentially composing animations with a relative offset. For example, to compose a first animation `anim1` with a second animation `anim2` which starts 0.5 seconds *before* the end of `anim1`.

```
relSeqAnim = relSequential anim1 anim2 (-0.5)
```

3.3 Expressiveness

In monadic DSLs the >>= and `return` combinators provide the needed expressivity. When creating inspectable animations, >>= is a liability since it has limited inspectability. PaSe supports extension with custom control flow combinators.

The `onlyDone` animation shows all *done* items while hiding all to-do items. This could be implemented by first showing all items with the `showAll` animation, since an item might have been hidden by a previous action, and then hiding all to-do items with the `hideToDo` animation. The definition is given below, while the definitions of `showAll` and `hideToDo` are omitted for brevity.

```
onlyDoneNaive = do showAll; hideToDo
```

However, we only intend to show completed items if needed. So instead we first check how many done items there are, if there are more than zero we play the previous version of `onlyDone`, otherwise we only hide the unfinished items.

```
onlyDone = do
  cond <- doneItemsGt0     -- check if more than 0 'done' items
  if cond then onlyDoneNaive else hideToDo
```

However, this formulation uses monadic features and is thus not inspectable. To make it inspectable, we utilize a custom combinator `ifThenElse`. We revisit this example in more detail in Section 5.

```
onlyDone = ifThenElse doneItemsGt0 onlyDoneNaive hideToDo
```

For this new combinator, we can define custom ways to inspect it. Since each branch might have a different duration, we do not choose to extract the duration but rather the *maximum* duration of the animation.

```
onlyDoneMaxDuration = maxDuration onlyDone -- = 1
```

Sections 2 and 3 gave a look and feel of the features of PaSe. In the following sections, we delve deeper into the internals of the implementation.

4 Implementation of PaSe

This section implements the previously introduced operations and redefines the animations to show the resulting type signature. We develop PaSe in the style of the `mtl` library[10] which implements monadic effects using type classes [10]. This style is also called the finally tagless approach [3]. However, because the PaSe classes are not subclasses of `Monad`, they leave room for inspectability.

4.1 Specifying Basic Animations

The `mtl` library uses type classes to declare the basic operations of an effect. Similarly, we specify the `linearTo` operation using the `LinearTo` type class.

```
class LinearTo obj f where
  linearTo :: Traversal' s Float -> Duration -> Target -> f ()
```

The traditional mtl style would add a `Monad f` superclass constraint. As it hinders inspectability, we defer the addition of this constraint to the user. This allows the definition of animations which are, for example `Applicative`, if inspectability is needed or `Monad` if it is not.

The `linearTo` function is used to specify basic animations like `line1Out`, `line2In`, `menuSlideIn`, and `appFadeOut` from Section 2. As an example, we redefine `line1Out` with its type signature; the others are similar.

```
line1Out :: (LinearTo Application f) => f ()
line1Out = linearTo (navbar . underline1 . width) (For 0.25) (To 0)
```

4.2 Specifying Composed Animations

Section 2 used the combinators `sequential` and `parallel` for composing animations. In this section, we describe these combinators in more detail.

Sequential Composition We reuse the `Functor`-`Applicative`-`Monad` hierarchy for sequencing animations.

ehe `liftA2` function from the `Applicative` class, which has type `Applicative f => (a -> b -> c) -> f a -> f b -> f c`, takes two animations `f a` and `f b` and returns a new animation which plays them in order. The final result of the animation is of type `c`, which is obtained by using the function `a -> b -> c` and applying the results of the two played animations to it.

The >>= function from the `Monad` class, which has type `Monad f => f a -> (a -> f b)`, takes an animation `f a` and then feeds the result of this animation into the function `a -> f b` to play the animation `f b`.

The `sequential` function is a specialization of the `liftA2` function. It only applies to animations with a `()` return value, and trivially combines the results.

[10] http://hackage.haskell.org/package/mtl

```
sequential :: (Applicative f) => f () -> f () -> f ()
sequential f1 f2 = liftA2 (\_ _ -> ()) f1 f2
```

Hence, the type signature for `selectBtn2Anim` contains an `(Applicative f)` constraint in addition to the `(LinearTo Application f)` constraint.

```
selectBtn2Anim :: (LinearTo Application f, Applicative f) => f ()
selectBtn2Anim = line1Out `sequential` line2In
```

Parallel Composition We create our own `Parallel` type class for the `parallel` function[11]. Its `liftP2` function has the same signature as `liftA2`, but the intended semantics of the `liftA2` implementation is parallel rather than sequential composition. Technically they are interchangeable, but the relation of `Applicative` to `Monad` makes it more sensible for sequential composition semantics. The `parallel` function is a specialization of `liftP2`.

```
class Parallel f where
  liftP2 :: (a -> b -> c) -> f a -> f b -> f c

parallel :: (Parallel f) => f () -> f () -> f ()
parallel f1 f2 = liftP2 (\_ _ -> ()) f1 f2
```

With that in place we can give a type signature for `menuIntro`.

```
menuIntro :: (LinearTo Application f, Parallel f) => f ()
menuIntro = menuSlideIn `parallel` appFadeOut
```

4.3 Running Animations

Now we create a new `Animation` data type that instantiates the above type classes to interpret PaSe programs as actual animations. We briefly summarize this implementation here and refer for more details to our codebase.[12]

The `Animation` data type, defined below, models an animation. It takes the current state `s` and the time elapsed since the previous frame. It produces a new state for the next frame, the remaining unused time and either the remainder of the animation or, if there is no remainder, the result of the animation. Note that the output is wrapped in a type constructor `m` to embed custom effects. We need the unused time when there is more time between frames than the animation uses. Then, the remaining time can be used to run the rest of the animation.

```
newtype Animation s m a = Animation { runAnimation ::
    s ->                              -- previous state
    Float ->                          -- time delta
    m ( s                             -- next state
      , Either (Animation s m a) a    -- remainder / result
      , Maybe Float )}                -- remaining delta time
```

[11] The `Alternative` class (https://en.wikibooks.org/wiki/Haskell/Alternative_and_MonadPlus) is not suitable as the laws are not the same.

[12] https://github.com/rubenpieters/anim_eff_dsl/tree/master/code

LinearTo Instance The `linearTo` implementation of `Animation` constructs the new state, calculates the remainder of the animation and the remaining delta time. The difference between the `linearTo` duration and the frame time determines whether there is a remaining `linearTo` animation or remaining time.

Examples We illustrate the behaviour on a tuple state (`Float`, `Float`), of an `x` and `y` value. The `right` animation transforms the `x` value to 50 over 1 second.

```
right :: (LinearTo (Float, Float) f) => f ()
right = linearTo x (For 1) (To 50)
```

We run it for 0.5 seconds by applying it to the `runAnimation` function, together with the initial state (`s0 = (0,0)`) and the duration `0.5`. We instantiate the `m` type constructor inside `Animation` with `Identity` as no additional effects are needed; this means that the result can be unwrapped with `runIdentity`.

```
(s1, remAn1, remDel1) = runIdentity (runAnimation right s0 0.5)
-- s1 = (25.0, 0.0) | remAn1 = Left anim2 | remDel1 = Nothing
```

Running `right` for 0.5 seconds uses all available time and yields the new state (`25, 0`). The remainder of the animation is the `right` animation with its duration reduced by `0.5`, or essentially `linearTo x (For 0.5) (To 50)`. Let us run this remainder for 1 second.

```
(s2, remAn2, remDel2) = runIdentity (runAnimation anim2 s1 1)
-- s2 = (50.0, 0.0) | remAn2 = Right () | remDel2 = Just 0.5
```

Now the final state is (`50, 0`) with result `()` and remaining time `0.5`.

Monad Instance For sequential animations we provide a `Monad` instance. Its `return` embeds the result `a` inside the `Animation` data type. The essence of the `f >>= k` case is straightforward: first, run the animation `f`, then pass its result to the continuation `k` and run that animation. We return the result of the animation, or, if there is an animation remainder, because the remaining time was used up, we return that remainder.

Examples Let us define an additional animation `up` which transforms the `y` value to 50 over a duration of 1 second. Additionally, we define an animation `rightThenUp` which composes the `right` and `up` animations in sequence.

```
up :: (LinearTo (Float, Float) f) => f ()
up = linearTo y (For 1) (To 50)

rightThenUp :: (LinearTo (Float, Float) f, Applicative f) => f ()
rightThenUp = right `sequential` up
```

Running the `rightThenUp` animation for 0.5 seconds gives a similar result to running `right` for 0.5 seconds. We obtain the new state `(25, 0)`, an animation remainder `anim2` and there is no remaining time. Now the animation remainder is the rest of `rightThenUp`, which is half of `right` and `up`. So, when we run this animation remainder for 1 second, it will run the second half of `right` and the first half of `up`. This results in the state `(50, 25)`, the animation remainder `anim3` and no remaining delta time. This animation remainder is of course the second half of the `up` animation. If we continue to run that remainder, for example for 1 second, then we get the final state `(50, 50)` and the animation result `()`.

Parallel Instance The `liftP2` implementation runs the animations `f1` and `f2` on the starting state. We match on the cases where `f1` and `f2` finish with a result or an animation remainder and remaining time. We check which of the animations have finished and repackage them either into a result or a new remainder, using the result combination function where appropriate. When the longest of the two parallel animations is finished while not fully using the remaining delta time, we continue running the remainder of the animation.

Examples Let us run the animations `right` and `up` in parallel, which means that both the `x` and `y` value will increase simultaneously.

```
rightAndUp :: (LinearTo (Float, Float) f, Parallel f) => f ()
rightAndUp = right `parallel` up
```

The result of running this animation for 0.5 seconds gives the state `(25, 25)` and no remaining time. If we continue the animation remainder we get the state `(50, 50)` and 0.5 seconds of remaining time.

4.4 Inspecting Animations

To inspect animations we instantiate them with `Const`. It wraps an `a` value and has a `b` phantom type parameter to trivially make it a functor.

```
newtype Const a b = Const { getConst :: a }
```

We might wonder why this extra work is necessary. After all, it is possible to obtain the duration of an animation by running the animation and keeping track of how long it takes. First, this is not an ideal approach for obtaining the duration. We might obtain erroneous results when doing this on conditional animations. Since only one branch of the conditional will be taken, while the other branch with a different duration might be taken in reality. Also, this approach is infeasible when there are effects embedded within the animation. Second, duration is one possible inspection target. Another example is tracking the used textures within an animation so they can be loaded automatically. For this to be possible we must run the inspection *before* the animation runs for the first time, since the textures must be loaded first.

Inspecting LinearTo To obtain the duration of a `linearTo` animation, we embed the duration in the `Const` wrapper.

```
instance LinearTo obj (Const Duration) where
  linearTo _ duration _ = Const duration
```

Inspecting Applicative It is not possible to inspect animations with a `Monad` constraint, but it is possible for animations with an `Applicative` constraint. The `Const` data type is not the culprit here, but rather the `>>=` method of the `Monad` class, which contains the limiting factor: a continuation function `a -> m b`.

Inspecting Parallel The duration of two parallel animations is the maximum of their durations. The `Par (Const Duration)` instance implements this.

```
instance Par (Const Duration) where
  liftP2 _ (Const x1) (Const x2) = Const (max x1 x2)
```

Examples The duration function is a specialization of the unwrapper function of the `Const` data type, namely `getConst`. We can feed our previously defined animations `selectBtn2Anim` and `menuIntro` from Section 2 to this function and obtain their durations as a result.

```
duration :: Const Duration a -> Duration
duration = getConst

selectBtn2AnimDuration :: Duration
selectBtn2AnimDuration = duration selectBtn2Anim -- = For 1.0

menuIntroDuration :: Duration
menuIntroDuration = duration menuIntro -- = For 0.5
```

When we try to retrieve the duration of a monadic animation, there is an error from the compiler: there is no `Monad` instance for `Const Duration`.

```
complicatedAnimDuration :: Duration
complicatedAnimDuration = duration complicatedAnim
-- No instance for (Monad (Const Duration))
```

4.5 Adding a Custom Operation

Custom operation are added by defining a corresponding class. For example, if we want to add a `set` operation, then we create the corresponding `Set` class.

```
class Set obj f where set :: Lens' obj a -> a -> f ()
```

Now, an animation using the `set` operation will incur a `Set` constraint.

```
checkIcon :: (Set CompleteIcon f, ...) => f ()
checkIcon = do ...; set (checkmark . color) green; ...
```

To inspect or run such an animation, we also need to provide instances for the `Animation` and `Const` data types. In the `Animation` instance, we alter the previous state by setting the value targeted by the `lens` to `a`. The duration of a `set` animation is 0, which is what is returned in the `Duration` instance.

```
instance (Applicative m) => Set obj (Animation obj m) where
  set lens a = Animation $ \obj t -> let
    newObj = Lens.set lens a obj
    in pure (newObj, Right (), Just t)

instance Set obj (Const Duration) where
  set _ _ = Const (For 0)
```

5 Interaction Between Inspectability and Expressivity

Haskell DSLs are typically monadic because the >>= combinator provides great expressive power. Yet, this power also hinders inspectability. This section shows how to balance expressiveness and inspectability with a custom combinator. This feature is *opt-in* in the sense that it is only required when inspectability is required. If that is no concern, then it is no problem to work with the `Monad` constraint.

Let us revisit the `onlyDone` animation from Section 3.3. The following definition imposes a `Monad` constraint on `f`, making the animation non-inspectable.

```
onlyDone :: (LinearTo Application f, Get Application f,
  Set Application f, Monad f, Parallel f) => f ()
onlyDone = do
  cond <- doneItemsGt0
  if cond then onlyDoneNaive else hideNotDone
```

However, there is duration-related information we can extract. For example, the *maximum duration* is the largest duration of the two branches.

To express this idea in PaSe we introduce an explicit combinator to replace this particular use of >>=, namely an `if-then-else` construction.

```
class IfThenElse f where
  ifThenElse :: f Bool -> f a -> f a -> f a
```

This is similar to the `handle` combinator from the `DynamicIdiom` class [21] and the `ifS` combinator from the `Selective` class [18].

Now we can reformulate `onlyDone` in terms of this `ifThenElse` combinator[13].

[13] Using GHC's `RebindableSyntax` extension, it is possible to use the builtin `if ... then ... else ...` syntax.

```
onlyDone :: (LinearTo Application f, Get Application f,
  Set Application f, Applicative f, Parallel f, IfThenElse f)
  => f ()
onlyDone = ifThenElse doneItemsGt0 onlyDoneNaive hideNotDone
```

We implement an appropriate `Animation` instance for `IfThenElse`.

```
instance (Monad f) => IfThenElse (Animation obj f) where
  ifThenElse fBool thenBranch elseBranch = do
    bool <- fBool
    if bool then thenBranch else elseBranch
```

Now, we can retrieve the maximum duration, using the `newtype` `MaxDuration` to signify this. The instance for `IfThenElse` retrieves the durations of the `then` and `else` branches and adds the greater value to the duration of the preceding animation inside the condition.

```
instance IfThenElse (Const MaxDuration) where
  ifThenElse (Const (MaxDur durCond)) (Const (MaxDur durThen))
             (Const (MaxDur durElse)) =
    Const (MaxDur (durCond + max durThen durElse))
```

This allows us to retrieve the maximum duration of the `onlyDone` animation.

```
onlyDoneMaxDuration :: MaxDuration
onlyDoneMaxDuration = maxDuration onlyDone -- = MaxDur 1.0
```

6 Interaction Between Callbacks and Inspectability

Many JavaScript animation libraries[14] exist, most of which allow the user to add custom behavior (which the library has not foreseen) through callbacks. A good example is the GreenSock Animation Platform (GSAP), a widely recommended and mature JavaScript animation library with a variety of features.

6.1 Working with GSAP

`TweenMax` objects are the GSAP counterpart of the `linearTo` operation. Their arguments are similar: the object to change, the duration, and the target value for the property. For example, animation `right` moves `box1` to the right:

```
const right = new TweenMax("#box1", 1, { x: 50 });
```

We can add animations to a `TimeLineMax` to create a sequential animation. Below, we create `rightThenDown` which moves `box1` to the right and then down.

[14] Examples: https://greensock.com, https://animejs.com, and https://popmotion.io.

```
const rightThenDown = new TimelineMax({ paused: true })
  .add(new TweenMax("#box1", 1, { x: 50 }))
  .add(new TweenMax("#box1", 1, { y: 50 }));
```

The `add` method takes the position on the timeline as an optional paramter. If we position both animations at point `0` on the timeline, they run in parallel. For example, the `both` animation below moves both `box1` and `box2` in parallel.

```
const both = new TimelineMax({ paused: true })
  .add(new TweenMax("#box1", 1, { x: 50 }), 0)
  .add(new TweenMax("#box2", 1, { x: 50 }), 0);
```

Timelines can also be embedded within other timelines.

```
const embedded = new TimelineMax({ paused: true })
  .add(both.play())
  .add(new TweenMax("#box1", 1, { y: 50 }), 0);
```

6.2 Callbacks and Inspectability

GSAP provides features related to inspectability. For example, we can use the `totalDuration` method to return the total duration of an animation. Ordinary animations correctly give their total duration when queried. For example, querying the duration of `embedded` correctly returns 2.

```
const embeddedDuration = embedded.totalDuration(); // = 2
```

However, if we want to provide animations similar to `onlyDone`, which contains an `if-then-else`, then the duration returned is not what we expect. The `add` method is overloaded and can also take a callback as parameter. Using the callback parameter we can embed arbitrary effects and control flow. For example, we can create a conditional animation `condAnim`, for which a duration of 1 is returned. This is because any callbacks that are added to the timeline are considered to have duration 0, even if an animation is played in that callback.

The resulting duration of 1 is different from the expected total duration of the animation, which is 2. Of course, in general the duration of the animations in both branches could differ, which is what makes it difficult to provide a procedure for calculating the duration of an animation in this form.

```
const condAnim = new TimelineMax({ paused: true })
  .add(both.play())
  .add(() => { if (cond) { new TweenMax("#box1", 1, { x: 50 }) }
               else { new TweenMax("#box2", 1, { x: 50 }) } });
const condAnimDuration = condAnim.totalDuration() // = 1
```

6.3 Relevance of Duration in Other Features

A wrongly calculated duration becomes more problematic when another feature relies on this calculation. The *relative sequencing* feature needs the duration of the first animation, so the second animation can be added with the correct offset. For example, we can specify the position parameter `-=0.5` to specify that it should start 0.5 seonds before the end of the previous animation.

```
const bothDelayed = new TimelineMax({ paused: true })
  .add(new TweenMax("#box1", 1, { x: 50 }), 0)
  .add(new TweenMax("#box2", 1, { x: 50 }), "-=0.5");
```

This feature differs from ordinary sequencing such as with `sequential`. When we state that animation `B` must play 0.5 seconds before the end of animation `A`, then it is not possible to wait until animation `A` has finished to start running animation B. This is because animation `B` *should have started playing for 0.5 seconds already.* When we have the duration of animation `A` available, animation `B` can be appropriately scheduled.

This feature behaves somewhat unexpectedly when combined with a conditional animation. In the `relativeCond` animation below we add a basic animation followed by a conditional animation. Then we add an animation with a relative position. The result is that the relative position is calculated with respect to the duration of the animations before it, which was a duration of 1.

```
const relativeCond = new TimelineMax({ paused: true })
  .add(new TweenMax("#box1", 1, { x: 50 }), 0)
  .add(() => { if (cond) { new TweenMax("#box1", 1, { x: 100 });
             } else { new TweenMax("#box1", 1, { x: 0 }); } })
  .add(new TweenMax("#box2", 1, { x: 50 }), "-=0.5");
```

Predicting the resulting behavior becomes much more complicated when conditional animations are embedded deep inside complex timelines and cause erroneous duration calculations. Clearly, being more explicit about control flow structures and their impact on inspectability like in PaSe helps providing a more predictable interaction between these features.

6.4 Relative Sequencing in PaSe

While not yet ideal from a usability perspective,[15] PaSe does enable correctly specifying relative sequential compositions by means of `relSequential`.

```
relSequential :: forall c g.
  (c (Const Duration), c g, Applicative g, Delay g) =>
  (forall f. c f => f ()) -> g () -> Float -> g ()
relSequential anim1 anim2 offset = let
  dur = getDuration (duration anim1)
  in anim1 `sequential` (delay (dur + offset) *> anim2)
```

[15] It requires `AllowAmbiguousTypes` (among other extensions) and explicitly instantiating the constraint `c` at the call-site.

Because this definition requires instances instantiated with `Const Duration`, it only works for animations whose duration can be analyzed. Now, we can correctly compose conditional animations sequentially using relative positioning. We use the `relMaxSequential` function to sequence animations with a maximum duration.

```
-- create synonym for multiple constraints
class (LinearTo Float f, IfThenElse f) => Combined f where
instance (LinearTo Float f, IfThenElse f) => Combined f where

relCond :: (LinearTo Float f, IfThenElse f, Applicative f) => f ()
relCond = relMaxSequential @Combined anim1 anim2 (-0.5)
```

7 Use Case

This section compares an implementation of a simplified Pac-Man game (Figure 5) in Haskell with PaSe and TypeScript with GSAP both quantitatively and qualitatively. The quantitative evaluation compares development time and lines of code. The qualitative one compares different aspects of the libraries.

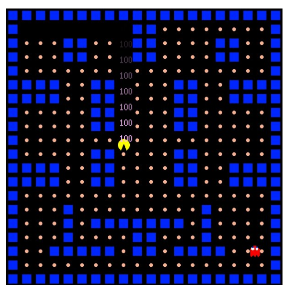


Fig. 5: Screenshot of the Pac-Man application.

7.1 Quantitative Evaluation

This section compares the PaSe and GSAP implementations on quantitative criteria. We consider the development time and lines of code for each module.

- **Development Time** The Haskell application was developed in ∼1.5 working days, while the TypeScript application took ∼1 working day. We consider this approximately the same development time as the Haskell application was developed first, and thus contains design work shared by both applications. The developer is proficient in both languages.
- **Lines of Code (LOC)** Table 1 contains the LOC data (including whitespace) for both applications. Their total LOCs are roughly the same. However, the Haskell code implements its own functionality for sprites and textures while we used the existing `Sprite` class of the `PixiJS` library in TypeScript.

Table 1: Lines of code comparison (including whitespace)

Module	Haskell/PaSe (LOC)	%	TypeScript/GSAP (LOC)	%
AnimDefs	127	21%	197	32%
Anims	43	7%	39	6%
Field	48	8%	77	12%
Game	130	21%	113	18%
Main	36	6%	23	4%
Sprite	45	7%	/	/
Textures	34	6%	13	2%
Types	10	2%	3	0%
View	139	23%	158	25%
Total	*612*	*100%*	*623*	*100%*

– **Relative LOC** Table 1 also contains the relative LOCs. The GSAP animation definitions (AnimDefs) are slightly bigger because we had to embed effects in the animations due to differences in the used graphics library, and because of TypeScript's relative verbosity. Using the timeline feature of GSAP, the code for simple animations is comparable to PaSe. However for more complex animations and those requiring embedded effects, there are more differences that we discuss in more detail in the qualitative evaluation.

7.2 Qualitative Evaluation

This section compares PaSe and GSAP on five qualitative criteria.

– **Eco-system** Animations are not created in isolation; they need to be coupled to a graphical backend to display them on the screen. GSAP's maturity makes it a clear winner here. It is well integrated with the browser and supports a rich set of features such as a variety of plugins, compatibility across browsers and support for animating a large range of DOM elements. Yet, for Pac-Man we only needed lenses for our own user-defined state.
– **Workflow** It is important that animations can be specified easily and concisely. Creating pure animations, without any embedded effects, are equally convenient in GSAP and in PaSe. However, more complex interactions with effects and control flow are simpler in PaSe. We saw this in the Pac-Man use case when implementing particle animations. A particle animation is an animation that creates an object, animates it and then destroys it again. We implemented a general wrapper for such animations which takes as input a function `Int -> Animation`, where the `Int` is the unique particle identifier, and a creation and deletion function for the particle. In the GSAP library we have to add the function to the timeline as a callback, which means its duration is considered to be `0`. This is problematic because the deletion of the particle should occur after its animation. This means that we are forced to manually calculate and provide the duration for the particle animation.

- **Performance** Both libraries perform equally acceptable on Pac-Man: no visible glitches or lag at 60 frames per second (FPS) on an Intel core i7-6600U at 2.60 GHz with 8GB memory. We have also implemented a benchmark similar to GSAP's speed test[16], which tests a large parallel animations. GSAP is slightly more optimized currently as it handles 500 parallel animations at 60 FPS instead of PaSe's 400. This could be remedied by further performance improvements of PaSe, like fusing multiple parallel animations or improving the `Animation` data structure, which are future work.
- **Extensibility & Inspectability** Extensibility and inspectability are key features of PaSe. Both were useful for Pac-Man. Inspectability allowed extracting all used textures in the animations to automate their loading. Extensibility enabled the definition of the particle effect mentioned earlier. We created a new `WithParticle` type class and implemented both an `Animation` instance and a `Const` instance for the texture inspection. GSAP does not support inspectability, and thus we did not implement the automatic loading of textures. The particle animation function was implemented with callbacks and implicit side-effects, which TypeScript allows anywhere.

8 Future Work

The main aspects to improve are currently related to the PaSe eco-system, such as providing bindings to various graphics backends, extending the provided set of features and improving the performance of the implementation.

Another avenue of future work is to explore trade-offs between the MTL style, as used in this paper, or an initial encoding approach, as is typical in approaches based on algebraic effects and handlers. The MTL style was chosen since it is simpler presentation-wise, mainly on the extensibility aspect with regards to different computation classes. However, we believe that implementation of some features, such as the relative sequencing, is simpler in the initial approach.

An aspect not touched in this paper is *conflict management*. A conflict appears when the same property is targeted by different animations in parallel. For example, if we want to change a value both to 0 and 100 in parallel, what should this animation look like? PaSe does no conflict management, and the animation might look stuttery. GSAP, for example, resolves this by only enabling the most recently added animation. However this strategy is not straightforwardly mapped to the context of PaSe. Inspectability could provide a solution for this problem by providing the possibility to detect conflicts.

9 Related Work

Functional Reactive Programming The origins of functional reactive programming (FRP) lie in the creation of animations [4], and many later developments use FRP as the basis for purely functional GUIs.

[16] https://greensock.com/js/speed.html

PaSe focuses on easily describing *micro-animations*, which differ from general *animations* as considered by FRP. The latter can typically be described by a time-paramterized picture function `Time -> Picture`. While a subset of all possible animations, micro-animations are not easily described by such a function because many small micro-animations can be active at the same time and their timing depends on user interaction.

We have only supplied an implementation of PaSe on top of a traditional event-based framework, but it is interesting future work to investigate an implementation of the `linearTo`, `sequential` and `parallel` operations in terms of FRP behaviours and events.

Animation Frameworks Typical micro-animation libraries for web applications (with CSS or JavaScript) and animation constructions in game engines provide a variety of configurable pre-made operations while composing complex animations or integrating new types of operations is difficult. PaSe focuses on the creation of complex sequences of events while still providing the ability to embed new animation primitives. We have looked at GSAP as an example of such libraries and some of the limits in combining extensibility with callbacks and inspectability. PaSe is an exercise in improving this combination of features forward in a direction which is more predictable for the user.

Planning-Based Animations PaSe shares similarities with approaches which specify an animation as a plan which needs to be executed [12,17]. An animation is specified by a series of steps to be executed, the plan of the animation. The coordinator, which manages and advances the animations, is implemented as part of the hosting application. PaSe realizes these plan-based animations with only a few core principles and features the possibility of adding custom operations and inspection. A detailed comparison with these approaches is difficult, since their works are very light on details of the actual implementation aspect.

Inspectable DSLs Some DSLs for parsing [9,2,15], non-determinism [11], remote execution [5,6] and build systems [19] focus on inspectability aspects, yet none of them provide extensibility and expressiveness in addition to inspection.

10 Conclusion

We have presented PaSe, an extensible and inspectable DSL for micro-animations. PaSe focuses on compositional animations using sequential and parallel animations as basic building blocks. This is in contrast with other animation libraries typically focused on sequential composition and callbacks with implicit effects.

We utilized a to-do list application use case to explain the features of PaSe. In this use case we showed the additional features of PaSe: extensiblity, inspectability and expressivity. We argue that the callback style of providing extensibility hurts the inspectability aspect of animations, which is found in for example the GreenSock Animation Platform. An implementation of the Pac-Man game confirms that this can be a problem even in simple applications.

References

1. Bederson, B.B., Boltman, A.: Does animation help users build mental maps of spatial information? In: INFOVIS 1999. pp. 28–35 (1999). https://doi.org/10.1109/INFVIS.1999.801854
2. Capriotti, P., Kaposi, A.: Free applicative functors. In: MSFP 2014. pp. 2–30 (2014). https://doi.org/10.4204/EPTCS.153.2
3. Carette, J., Kiselyov, O., Shan, C.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. J. Funct. Program. **19**(5), 509–543 (2009). https://doi.org/10.1017/S0956796809007205
4. Elliott, C., Hudak, P.: Functional reactive animation. In: ICFP 1997. pp. 263–273 (1997). https://doi.org/10.1145/258948.258973
5. Gibbons, J.: Free delivery (functional pearl). In: Haskell 2016. pp. 45–50 (2016). https://doi.org/10.1145/2976002.2976005
6. Gill, A., Sculthorpe, N., Dawson, J., Eskilson, A., Farmer, A., Grebe, M., Rosenbluth, J., Scott, R., Stanton, J.: The remote monad design pattern. In: Haskell 2015. pp. 59–70 (2015). https://doi.org/10.1145/2804302.2804311
7. Gonzalez, C.: Does animation in user interfaces improve decision making? In: CHI 1996. pp. 27–34 (1996). https://doi.org/10.1145/238386.238396
8. Heer, J., Robertson, G.G.: Animated transitions in statistical data graphics. IEEE Trans. Vis. Comput. Graph. **13**(6), 1240–1247 (2007). https://doi.org/10.1109/TVCG.2007.70539
9. Hughes, J.: Generalising monads to arrows. Sci. Comput. Program. **37**(1-3), 67–111 (2000). https://doi.org/10.1016/S0167-6423(99)00023-4
10. Jones, M.P.: Functional programming with overloading and higher-order polymorphism. In: Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text. pp. 97–136 (1995). https://doi.org/10.1007/3-540-59451-5_4
11. Kiselyov, O.: Effects without monads: Non-determinism - back to the meta language. In: ML/OCaml 2017. pp. 15–40 (2017). https://doi.org/10.4204/EPTCS.294.2
12. Kurlander, D., Ling, D.T.: Planning-based control of interface animation. In: CHI 1995. pp. 472–479 (1995). https://doi.org/10.1145/223904.223968
13. van Laarhoven, T.: CPS-Based Functional References (2009), https://www.twanvl.nl/blog/haskell/cps-functional-references
14. Liang, S., Hudak, P., Jones, M.P.: Monad transformers and modular interpreters. In: POPL 1995. pp. 333–343 (1995). https://doi.org/10.1145/199448.199528
15. Lindley, S.: Algebraic effects and effect handlers for idioms and arrows. In: WGP 2014. pp. 47–58 (2014). https://doi.org/10.1145/2633628.2633636
16. McBride, C., Paterson, R.: Applicative programming with effects. J. Funct. Program. **18**(1), 1–13 (2008). https://doi.org/10.1017/S0956796807006326
17. Mirlacher, T., Palanque, P.A., Bernhaupt, R.: Engineering animations in user interfaces. In: EICS 2012. pp. 111–120 (2012). https://doi.org/10.1145/2305484.2305504
18. Mokhov, A., Lukyanov, G., Marlow, S., Dimino, J.: Selective applicative functors. ICFP 2019 pp. 90:1–90:29 (2019). https://doi.org/10.1145/3341694
19. Mokhov, A., Mitchell, N., Peyton Jones, S.: Build systems à la carte. PACMPL **2**(ICFP 2018), 79:1–79:29 (2018). https://doi.org/10.1145/3236774
20. Wadler, P.: Comprehending monads. In: LFP 1990. pp. 61–78 (1990). https://doi.org/10.1145/91556.91592

21. Yallop, J.: Abstraction for web programming. Ph.D. thesis, University of Edinburgh, UK (2010)