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

Abstract. This paper presents PaSe, an extensible and inspectable DSL
embedded in Haskell for expressing micro-animations. The philosophy of
PaSe is to compose animations based on sequential and parallel compo-
sition of smaller animations. This differs from other animation libraries
that focus more on sequential composition and have only limited forms
of parallel composition. To provide similar flexibility as other animation
libraries, PaSe features extensibility of operations and inspectability of
animations. We present the features of PaSe with a to-do list applica-
tion, discuss the PaSe implementation, and argue that the callback style
of extensibility is detrimental for correctly combining PaSe features. We
contrast with the GreenSock Animation Platform, a professional-grade
and widely used JavaScript animation library, to illustrate this point.

1 Introduction

Monads quickly became ubiquitous in functional programming because of their
ability to structure effectful code in a pure functional codebase [22]. However,
monads have two major drawbacks. First, monads are not trivially extensible. A
variety of techniques were developed to resolve this, including monad transform-
ers [14], free monads, and algebraic effects and handlers [21]. Second, monadic
computations can only be inspected up to the next action. Techniques such as
applicative functors [16], arrows [9], or selective applicative functors [18] increase
the inspection capabilities by reducing the expressivity compared to monads.

This paper develops a domain specific language (DSL) embedded in Haskell
for defining micro-animations, called PaSe[1]. PaSe employs the aforementioned
techniques to support its key features: extensibility of operations and inspectabil-
ity of animations while providing the freedom to express arbitrary animations.

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 applica-
tion [1,7,8]. Examples can be found in almost every software application: window
managers shrink minimized windows and move them towards the taskbar, menus

---

[1] Pronounced *pace*, the name is derived from **pa**rallel and **se**quential.

in mobile applications pop in gradually, browsers highlight newly selected tabs with an animation, etc.

This paper develops a DSL which combines various features expected of animation libraries, by building them on top of recent ideas found in functional programming. More concretely, our contributions are as follows:

- We develop PaSe which supports correct interactions for expressing arbitrary animations and inspectability. Animation libraries, such as the GreenSock Animation Platform (GSAP)[2], typically use callbacks as a means of extensibility/expressivity which is detrimental to inspectability. We show an example where this results in unexpected behaviour, and how to correct it.
- We extend and develop PaSe as an example of an *extensible* DSL, in the sense that operations can be freely added. While there is a variety of literature on approaches to extensibility, there is not much literature devoted to use cases.
- Animations in PaSe can be specified in an *inspectable* manner, in the sense that information is obtained from a computation without running it. While inspectability has been studied for specific types of computations, such as free applicative functors [2], it is a novel aspect to combine it with extensibility.
- PaSe contains a primitive for parallel animation composition. This composition form can be arbitrarily nested and embedded anywhere within an animation and interacts correctly with the other features of PaSe: extensibility and inspectability. Parallel components in sequentially specified components is not new, see for example the *par* element in UML sequence diagrams [20], or the parallel statement in Ren'Py[3] or React Native Animations[4]. But, those systems either do not provide a form of parallelism as general or interaction with other features such as inspectability is not present.

2   Motivation

This section presents a simplified to-do list application to showcase the functionality and features of PaSe.

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.

In this application, various user actions are accompanied with an animation:

- Each item can be marked as *done* or *not done* by clicking on it. The checkmark icon changes shape and color to indicate the change in status. These are the markAsDone/markAsToDo animations, of which markAsDone is shown in Figure 2a.

---

[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

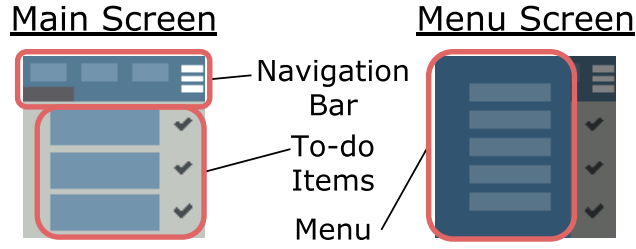## Main Screen   Menu Screen

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

- Items can be filtered by their status by using the navigation bar buttons. The first button shows all items, the second shows all done items, and the third shows all to-do items. Both the navigation bar underline and the to-do items itself change shape to indicate the change in selection. These are the showAll/onlyDone/onlyToDo animations, of which onlyDone is shown in Figure 2b.
- The menu screen shows/hides itself after clicking the hamburger icon. The menu expands inward from the left, to indicate the change in application state. These are the menuIntro/menuOutro animations, of which menuIntro is shown in Figure 2c.
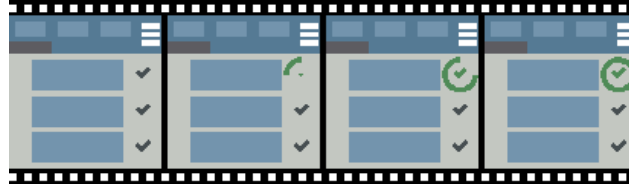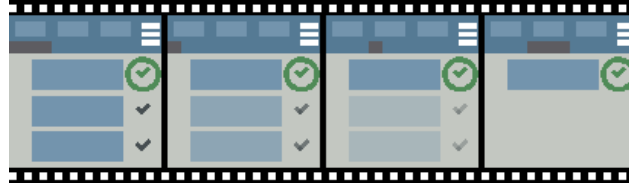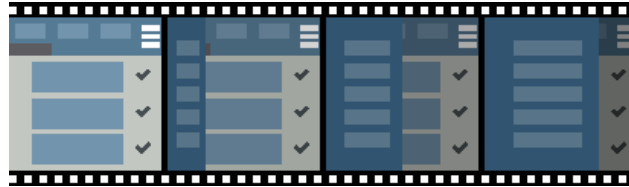

(a) markAsDone: the checkmark changes shape and color.

(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.

2.2   Composing Animations

We express animations in PaSe by composing smaller animations into larger ones. When creating an animation, we need come up with a suitable decomposition. 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.

2.3   Basic Animations

A basic animation changes the property of an element over a period of time. To specify a basic animation we need three elements: a lens specifying which property to change, the duration of the animation, and lastly the target value for this property. Since this operation creates an animation which changes a property *to* a target linearly, it is called `linearTo`. The duration is specified with the `For` constructor while the target value is specified with the `To` constructor.

Note on Lenses We use the lens notation `x . y . z` to target the property `z` inside a nested structure `{ x: { y: { z: Property } } }`. This type of lenses was conceived by van Laarhoven [13], and later packaged into various Haskell libraries, such as `lens`[5].

To implement the navigation bar underline animation, we reduce the underline width of the first button for 0.25 seconds and increase the underline width of the second button for 0.25 seconds. These animations are expressed in respectively `line1Out` and `line2In` below, and shown visually in Figure 3.

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

Other examples are the `menuSlideIn` and `appFadeOut` animations. For the former, we increase the width of the menu over a duration of 0.5 seconds, and for the latter we increase the alpha value of the obscuring box over a duration of 0.5 seconds. These animations are shown visually in Figure 3.

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

2.4   Composed Animations

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

To obtain the `selectBtn2` animation, we combine both the `line1Out` and `line2In` animations with the `sequential` combinator. This constructs a new animation which first plays the `line1Out` animation, and once it is finished plays the `line2In` animation.

---

[5] https://hackage.haskell.org/package/lens
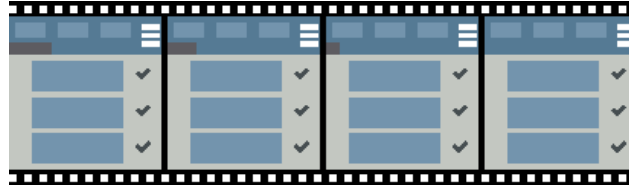
(a) The `line1Out` animation.
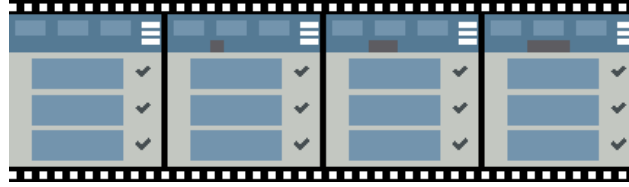
(b) The `line2In` animation.
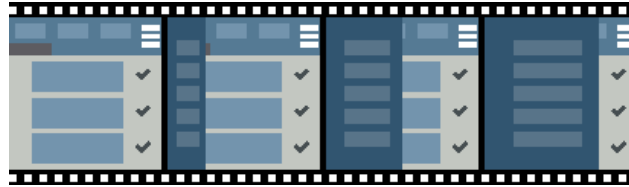
(c) The `menuSlideIn` animation.
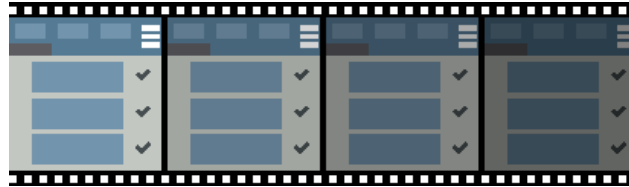
(d) The `appFadeOut` animation.

Fig. 3: Basic `linearTo` animations.

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

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

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

Both of these animations are shown visually in Figure 4.

3   Extensibility, Inspectability and Expressiveness

The features seen in Section 2 form the basis of PaSe. However, animation libraries such as GSAP provide a much more extensive list of features.

(a) The `selectBtn2` animation.
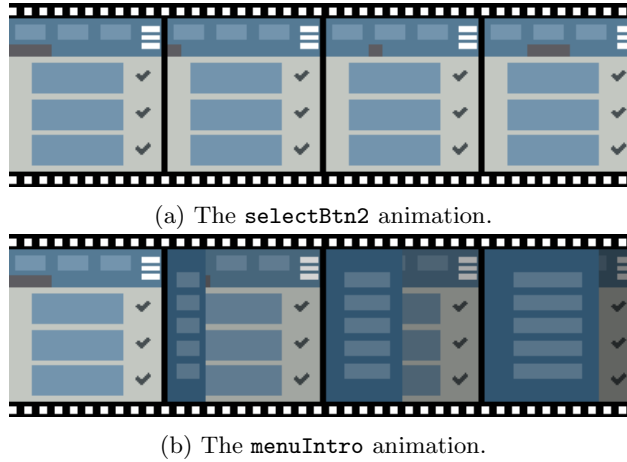

(b) The `menuIntro` animation.

Fig. 4: All of the defined composed animations.

To add support for similar features, we design PaSe with extensibility and inspectability in mind. This means that PaSe can be extended with custom 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 custom combinators.

3.1 Extensibility

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

An example in the to-do list app is the `checkIcon` animation, part of the `markAsDone` animation, 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, if we want to know the duration of `menuIntro` without actually running it and keeping track of the time. We can do this by using a predefined `duration` function, which calculates the duration by inspecting the animation. This 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 of an animation, then we cannot know upfront how long this animation will last. Requesting to calculate the duration of such an animation results in a type error.

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

Calculating a duration is mostly a stepping stone towards supporting 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

Similarly to providing custom operations, PaSe also supports custom combinators. For typical monadic DSLs this is not needed as `>>=` and `return` cover all use cases. However, since PaSe has the additional requirement of being inspectable, the `>>=` combinator ends up being a liability because it only provides a very limited amount of inspectability.

In the `onlyDone` animation, we show 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 for this is given below, while the implementation of `showAll` and `hideToDo` is omitted.

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

However, this animation is a bit naive since it executes the `showAll` animation regardless of whether there are any hidden done items that actually need to appear. Instead, the intended animation is to only make done items appear when any were hidden. This is done by first checking whether there are any done items and based on that we play the naive version of `onlyDone`, otherwise we just hide the not done items.

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

However, if we also want to inspect this computation, this formulation is problematic since it is specified using a construction which is too general; we explain this in more detail in Section 5. Instead, we can define a custom combinator `ifThenElse` to express `onlyDone` in an inspectable manner.

```
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 earlier introduced operations and redefines the animations to show the resulting type signature. We develop PaSe in the style of the `mtl` library[6] which implements monadic effects using typeclasses [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 typeclasses to declare the basic operations of an effect. Similarly, we specify the `linearTo` operation using the `LinearTo` typeclass.

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

The traditional mtl-style would add a `Monad f` superclass constraint. Yet, because it hinders inspectability, we do not require it everywhere. Instead, while we can still choose to add an additional `Monad f` constraint in selected cases, we leave the freedom to opt for an alternative like `Applicative f` in other cases.

The `linearTo` function of the `LinearTo` typeclass specifies basic animations like `line1Out`, `line2In`, `menuSlideIn`, and `appFadeOut` animations 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. Here, we provide these as well as more general combinators.

Sequential Composition We reuse the `Functor`-`Applicative`-`Monad` hierarchy for sequencing animations. In particular, the `sequential` function is a specialization of the `liftA2` function[7].

```
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 The `parallel` function does not correspond to a member function in an existing Haskell typeclass. At first we might consider the `<|>` combinator in the `Alternative` class, but there is no sensible corresponding `empty` animation; also the suggested laws[8] do not make sense for animations.

Instead, we create our own `Parallel` typeclass below. Its function `liftP2` has the same signature as `liftA2`, but denotes parallel composition instead of sequential composition. 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
```

and Now 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` datatype that instantiates the above typeclasses to interpret PaSe programs as actual animations. We briefly summarize this implementation here and refer for more details to our prototype implementation.[9]

The function type `obj -> Float -> m (obj, (Either (Animation obj a) a))` models an animation. It takes as inputs the the current application state and the time elapsed since the previous frame. As output it produces a new application state for the next frame together with 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. One more detail

---

[7] https://hackage.haskell.org/package/base-4.12.0.0/docs/Control-Applicative.html#v:liftA2

[8] https://en.wikibooks.org/wiki/Haskell/Alternative_and_MonadPlus

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

is missing in the output: the remaining unused time. We need this when, for example, there is more time between frames than the animation uses. Then, the remaining time can be used to run part of the next animation. All together, we get the following type definition.

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

LinearTo Instance The `linearTo` implementation of `Animation` does three things: construct the new object state, calculate the remainder of the animation, and calculate the unused time. The difference between the `linearTo` duration and the frame time determines whether there is a remaining `linearTo` animation or remaining time.

Examples Let us illustrate the behaviour on a simple application state that is a tuple, `type State = (Float, Float)`, of an x-value and a y-value. The `right` animation transforms the x-value to 50 over a time of 1 second.

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

We run the animation for 0.5 seconds by applying the `runAnimation` function on `right`, together with the initial state (`s0 = (0,0)`) and the duration 0.5. We also instantiate the m type constructor m inside the `Animation` data type 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 animation finishes in state `(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. Of course, running `f` does not necessarily give us a result `a`. Running an animation gives us either a result or an animation remainder; we also have to take into account the potential remaining time. When we have an animation remainder instead of a result, we need to repackage this remaining animation as `remainder >>= k` instead of using `k a`. Additionally, if there is no remaining time, we do not continue running the continuation and instead return the current state and animation 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 State f) => f ()
up = linearTo y (For 1) (To 50)

rightThenUp :: (LinearTo State 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 the `rightThenUp` animation, which is half of the `right` animation and the full `up` animation. 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 For parallel animations the `liftP2` implementation runs the animations `f1` and `f2` both on the starting state. We distinguish different cases depending on whether `f1` and `f2` finish with a result or an animation remainder, and the 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 State 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

Inspecting an animation is done by interpreting PaSe to a different data type, which provides instances for the typeclasses with different semantics. In particular we show how to inspect the duration of an animation by means of the `Const` functor.

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

Inspecting LinearTo To inspect animations we provide an instance for the operations which interprets them into `Const Duration`. In the case of the `linearTo` operation, we simply embed the duration within a `Const` wrapper.

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

Inspecting Applicative While the `Const` data type does not provide a `Monad` instance, it does provide an `Applicative` instance. This means that 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 a continuation function `a -> m b`, which is the limiting factor of inspection.

Inspecting Parallel The duration of two parallel animations is the maximum of their durations. The `Par` instance implementation for `Const Duration` captures 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 the duration of the animations 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 an animation with a monad constraint, we receive an error from the compiler: it cannot find a `Monad` instance for `Const Duration`.

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

4.5   Adding a Custom Operation

Adding a custom operation is done as in any other mtl-style approach: by defining a new class for this operation. For example, if we want to add a `set` operation, then we create a corresponding `Set` class.

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

Now, an animation that uses 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)
```

This section gave an overview of the features provided by PaSe, however combining inspectability while allowing freedom of expressivity is not always straightforward. Therefore, we look at an example of the interaction of these two features in the next section.

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.

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, it does seem like we should be able to extract some duration related information from it. For example, the maximum duration should be the largest duration of the two branches.

To express this idea in PaSe we introduce an explicit combinator to replace the 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 [23] and the `ifS` combinator from the `Selective` class [18].

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

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

Not much changes for `Animation`, which implements `ifThenElse` in terms of the >>= combinator.

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

What is new is that we can now also retrieve the maximum duration, using a new type `MaxDuration` to signify that we are not simply calculating the duration of the animation. 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[10] 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

GSAP provides primitives similar to the `linearTo` and `sequential` operations in PaSe. While it does not have a primitive for `parallel` composition, we can define it in terms of other GSAP features.

`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 });
```

A sequential animation is created by creating a timeline and `add`ing animations to it. Below, we create the animation `rightThenDown` which moves `box1` to the right and then moves it down.

```
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. Hence, if we position both animations at point `0` on the timeline, they will run in parallel. For example, the `both` animation below moves both `box1` and `box2` in parallel to an `x`-value of 50.

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

Since we have specified the option `paused` with `true`, we must call the `play` method to run an animation, as in `both.play()`. Using this behaviour, and the ability to nest timelines, we can provide the PaSe primitives.

```
function sequential(tl1, tl2) {
  return new TimelineMax({ paused: true })
    .add(tl1.play())
    .add(tl2.play()); }

function parallel(tl1, tl2) {
  return new TimelineMax({ paused: true })
    .add(tl1.play(), 0)
    .add(tl2.play(), 0); }
```

---

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

6.2   Callbacks and Inspectability

GSAP also provides features related to inspectability. For example, we can use the `totalDuration` method to return the total duration of an animation. Animations created with the previously defined `sequential` and `parallel` functions correctly give their total duration when queried. For example, we define an animation which moves both boxes in parallel to `x = 50` and then back to `x = 0` in parallel. The duration is correctly returned as 2.

```
const bothTo50 = parallel(
  new TweenMax("#box1", 1, { x: 50 }),
  new TweenMax("#box2", 1, { x: 50 }));

const bothTo0 = parallel(
  new TweenMax("#box1", 1, { x: 0 }),
  new TweenMax("#box2", 1, { x: 0 }));

const bothAnimation = sequential(bothTo50, bothTo0);
const bothAnimDuration = bothAnimation.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 to take a callback as parameter, which we used in the previous section since `tl.play()` is shorthand for the callback `() => tl.play()`. 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 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(bothTo50.play())
  .add(() => { if (cond) { bothTo0.play()   }
              else      { bothTo100.play() } });
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 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 more predictable interaction between these features.

6.4   Relative Sequential Composition in PaSe

While not yet ideal from a usability perspective,[11] 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)
```

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.

```
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)
```

---

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

7   Future Work

The current status of PaSe is a conceptual design in the space of animation libraries. A logical next step is to put it to the test and aim for the implementation of features provided by a mature animation library such as GSAP.

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 that was not touched in this paper is *conflict management*. A conflict appears when the same two properties are targeted by different animations in parallel. For example, if we want to change a certain value both to 0 and 100 in parallel, how 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.

8   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.

PaSe focuses on easily describing *micro-animations*, which differ from general *animoations* 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.

9   Conclusion

We have presented PaSe, an extensible and inspectable DSL for micro-animations. PaSe focuses heavily on both sequential and parallel composition of animations. This is in contrast with other animation libraries which are mostly focused on sequential composition and only provide a limited form of parallel composition.

We explained the features of PaSe with the help of a to-do list application use case. In this use case we showed the use of 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.

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. Object Management Group: Unified Modeling Language 2.5.1 (2017), https://www.omg.org/spec/UML/2.5.1
21. Plotkin, G.D., Pretnar, M.: Handlers of algebraic effects. In: ESOP 2009. pp. 80–94 (2009). https://doi.org/10.1007/978-3-642-00590-9_7
22. Wadler, P.: Comprehending monads. In: LFP 1990. pp. 61–78 (1990). https://doi.org/10.1145/91556.91592
23. Yallop, J.: Abstraction for web programming. Ph.D. thesis, University of Edinburgh, UK (2010)