Efficient Functional Reactive Programming through Incremental Behaviors

Bob Reynders and Dominique Devriese

imec - DistriNet, KU Leuven
{firstname.lastname}@cs.kuleuven.be

Abstract. Many types of software are inherently event-driven ranging from web applications to embedded devices and traditionally, such applications are implemented using imperative callbacks. An alternative approach to writing such programs is functional reactive programming (FRP). FRP offers abstractions to make event-driven programming convenient, safe and composable, but they come at a price. FRP behaviors cannot efficiently deal with larger, incrementally constructed values such as a collection of messages or a list of connected devices. Since these situations occur naturally, it hinders the use of FRP. We report on a new FRP primitive: 'incremental behavior'. We show that the semantics fit within existing FRP semantics and that their API can be used as a foundation for more ad-hoc solutions, such as incremental collections and discrete behaviors. Finally, we present benchmarks that demonstrate the advantages of incremental behaviors in terms of reduced computation time and bandwidth.

1 Introduction

Event-driven applications are common in several domains. Traditionally, such applications are implemented using imperative callbacks. An alternative approach to writing such programs is functional reactive programming (FRP). It offers abstractions to make event-driven programming convenient, safe and composable. It has been successfully applied to both GUI programming [5], embedded devices [23], etc.

FRP semantics define two primitives: events (a stream of values at discrete times) and behaviors (time-varying values). Let us introduce these with a small example, an FRP equivalent for the common case of using event handlers to increase a mutable sum:

```
val ints: Event[Int]   = ...                              //   3, 5, 2, ...
val sum: Behavior[Int] = ints.fold†(0) { (x, y) ⇒ x + y } // 0, 3, 8, 10, ...
```

We assume the existence of `ints`, an event that contains integers. We use the `fold`† method on events to build up state.[1] It takes an initial value (`0`) and an

[1] `fold`† is marked with † for clarity since a variant named `fold` is introduced in Section 2.

accumulation function (`(x, y) ⇒ x + y`) as arguments and builds a behavior. The event's values are accumulated starting with the initial value.

An FRP application is constructed by composing *behaviors* and *events* with a set of FRP operations. It typically defines a main behavior to describe the entire application, for example `Behavior[UI]` as the main value for a GUI application.

While FRP is nice in theory, there are shortcomings that crop up when you use it in practice. This paper focuses on one of those issues.

Computational Overhead. A practical problem with FRP is that behaviors containing large incrementally constructed values often behave suboptimally, for example a chat view:

```
val msgs: Event[Message] = ...
val chat: Behavior[List[Message]] = msgs.fold†(List.empty[Message]) { (lst, m) ⇒ m :: lst }
val chatView: Behavior[List[String]] = chat.map(_.map(_.pretty))
```

From an event stream of messages (`msgs`) we accumulate the state of the program (`chat`). All the messages are concatenated into a list behavior. A view of the state is generated through `map` by pretty printing all elements.

The problem here is that FRP only keeps track of the complete values within behaviors. It does not keep track of *how* it changes. In the example above, this means that a change to `chat` (through `msgs`) is propagated to `chatView` as 'there is a new list'. The entire list in the view is then re-mapped every time a new message is added. This makes the occurrence of a new message take $\mathcal{O}(n)$ processing time instead of a possible $\mathcal{O}(1)$.

This is especially problematic since maintaining large collections in behaviors is common in lots of FRP applications: chat applications have a list of messages, social networks have news feeds, sensor networks have lists of nodes, etc. In practice this means that FRP programmers work around the problem by using events to model concepts that would fit a behavior better, such as representing the chat view not as a behavior, but as an event of added strings.

Bandwidth Overhead. Computational complexity is not the only area in which standard behaviors do not perform optimally. Bandwidth intensive operations such as saving a behavior's history to disk (for logging or debugging purposes) or sending its data across the network, are directly impacted by knowing *how* behaviors change. The multi-tier FRP-based language as proposed in [19] is a typical example where bandwidth matters. To demonstrate this problem in our chat example, we extend it to continuously broadcast to clients:

```
def broadcastToClients(b: Behavior[List[String]]) = ...
broadcastToClients(chatView)
```

In this case, there is no way to efficiently implement `broadcastToClients` since behaviors cannot express *when* or *how* values update. Its only options are to poll for changes followed by either recomputing and transmitting the differences, or by sending the entire new list. In practice this often means that functions similar to `broadcastToClients` are modeled with less appropriate abstractions such as events:

```
def broadcastToClients(init: List[String], changes: Event[List[String]])
```

In addition to reducing computational and bandwidth overhead there are other reasons to express *how* behaviors change. For example, in an FRP Html library the interface may be modeled as a `Behavior[Element]`. Compared to completely rewriting the DOM, it would be much more efficient to apply only the changes of such a behavior.

1.1 Contributions

To summarize, we make the following contributions:

- We define incremental behaviors and their API and show how they fit within existing FRP semantics.
- We show how our approach is more general than previous work such as incremental collections [14, 17] and discrete behaviors [15, 20, 3] by implementing them into our framework. Additionally, we show how a joint API between discrete and incremental behaviors based on manually computing differences can form a middle ground between them.
- We present an implementation of incremental behaviors and incremental collections as a Scala library.[2] We demonstrate the advantages of incremental behaviors through a performance analysis of our implementation. In the analysis we compare incremental behavior's computational and bandwidth overhead with their non-incremental counterpart.

We start by introducing FRP and incremental behaviors in Section 2, and we show how incremental collections and discrete behaviors can be implemented on top of their API in Section 3. In Section 4, we evaluate their performance. Section 5 discusses incremental behaviors with respect to higher-order FRP semantics. We highlight related work in Section 6 and conclude with future work in Section 7.

In addition to our own implementation, we also found an independent implementation of similar ideas in the grapefruit-frp Haskell library [12] (their incremental signals seem similar to our incremental behaviors). This implementation has not been presented in the literature and lacks some of the features described here, but we consider it as additional evidence of the value of incremental behaviors.

2 Incremental Behaviors

We present incremental behaviors, an additional primitive for functional reactive programming (FRP). All code examples use our working proof-of-concept Scala implementation and we encourage the reader to play around with it.[2] As a small Scala introduction, in this paper it is sufficient to think of a `trait` as a Java-like

[2] Url omitted to stay anonymous, an archive of the project has been added to the submission.

```
trait Event[A] {
  def map[B](f: A ⇒ B): Event[B]
  def filter(p: A ⇒ Boolean): Event[A]
  def merge(e: Event[A])(f: (A, A) ⇒ A): Event[A]
  def fold†[B](init: B)(accum: (B, A) ⇒ B): Behavior[B]
}

trait Behavior[A] {
  def map2[B, C](b: Behavior[B])(f: (A, B) ⇒ C): Behavior[C]
  def map[B](f: A ⇒ B): Behavior[B]
  def snapshot[B, C](e: Event[B])(f: (A, B) ⇒ C): Event[C]
}
object Behavior { def constant[A]: Behavior[A] }
```

Fig. 1: Event & Behavior API

interface, `object Foo` as a collection of static methods for `Foo` and the `case class A(x: Int)` as a data type with field `x`. Case classes can get constructed (like regular classes) through either `new A(0)` or just `A(0)`.

2.1 Functional Reactive Programming: Event & Behavior

We begin with a summary of FRP and its semantics. We focus on first-order FRP semantics that are very similar to the ones defined in [11, 3]. For readers interested in higher-order FRP semantics we refer to Section 5. Let us go over the two main FRP primitives, event and behavior:

Events are sets of discrete values:

$$[\![Event_\tau]\!] = \{e \in \mathcal{P}(Time \times [\![\tau]\!]) \mid \forall (t, v), (t', v') \in e.\, t = t' \Rightarrow v = v'\}$$

In the denotational semantics above $[\![\alpha]\!]$ is the 'denotation' or *meaning* of α, $\{e \in \mathcal{P}(\alpha) \mid P\}$ is the set of elements e from the powerset of α for which P holds. Events are sets of ($Time$, $[\![\tau]\!]$) tuples that do not contain values with duplicate $Time$ components.

Typical examples of these discrete values are mouse clicks or button presses. There are three core operations: `map`, `filter` and `merge` as shown in fig. 1. We do not discuss `map` or `filter` since they behave just like their well-known collection counterparts. `merge` takes two events and returns an event that fires whenever one of the original events fire. When both fire at the same time, the given function combines both values into a single new one.

Behaviors can be thought of as values that can vary continuously over time. Semantically, behaviors of type τ are regular functions from $Time$ to τ:

$$[\![Behavior_\tau]\!] = \{b \in Time \to [\![\tau]\!]\}$$

An example of a behavior is the cursor's position. A mouse is always somewhere but its position may change continuously as you move your hand. The two

```
case class Entry(title: String, content: String) { val pretty = s"$title\n$content" }

val submissionE: Event[Entry] = ...

val todoListB: Behavior[List[Message]] =
  submissionE.fold†(List.empty) { (lst, msg) ⇒ msg :: lst }
val todoListView: Behavior[String] = todoListB.map(_.map(_.pretty).mkString)

def replicate(b: Behavior[String]) = ...
replicate(todoListView)
```

Fig. 2: FRP Todo List Example

core operations on behaviors are: `map2` and `constant` as shown in fig. 1. `constant` creates a behavior that never changes its value. `map2` has the ability to combine two behaviors with a function. Other convenience functions such as `map` can be defined in terms of `constant` and `map2`.

Behaviors ⇔ Events. Converting from events to behaviors and vice versa is done through two other operations: `Event.fold`†[3] and `Behavior.snapshot`, also shown in fig. 1. Folding an event is similar to folding a list, a starting value and an accumulation function is given to compute a new value whenever a new element arises. Its result is a behavior representing the accumulation. Snapshotting a behavior with an event inspects the value of a behavior at the rate of that event. The behavior is sampled for every change in the event by applying a combination function to the event value and the behavior's value at the time.

The FRP semantics that we just showed make a couple of design decisions that can differ from others: it is first-order instead of higher-order (see Section 5), it allows only one event value at a time and behaviors are defined in continuous time opposed to discrete time (see discrete behaviors in Section 3.2).

2.2 Motivating Example: Todo List

An example FRP program using our Scala library is shown in fig. 2. We implement a simple todo list. We leave out most of the code and focus only on the bits that are important to this paper. The user's intent to submit his message is modeled by the `submissionE` event. The state of the todo application itself is created by accumulating all the submissions into a list behavior (`todoListB`).

We create `todoListView`, a string representation of all the entries in the list by first turning the list of entries into a pretty printed list of strings (`_.map(_.pretty)`) and then concatenating all elements with `.mkString`.

Without going into details of its implementation, we assume the `replicate` function that takes a behavior and replicates it to a different application, such as in client/server applications.

[3] Note that to have a definable semantics for *fold* an extra restriction on events (which we omitted for brevity) is required. The occurrences in an event must be 'uniform discrete', that is, the amount of events before any time t must be finite.

This example demonstrates the computation and bandwidth issues for large values that we discussed before. Each new submission accumulates into the application's state (`todoListB`), and the mapping to `todoListView` always recomputes the entire pretty printed string since `todoListB` does not contain information about *how* it changes. Furthermore, the `replicate` function is impossible to implement efficiently since the newly created `todoListView` behavior does not contain information about *how* it changes either. `persist` has to detect changes and either recompute the differences between two behavior values, or send the entire behavior's state.

Depending on the amount of submissions, both problems can impact the user experience. A programmer has to either accept an underperforming application or remodel his code with less appropriate abstractions such as events as a workaround.

2.3 Incremental Behaviors

The purpose of our new FRP primitive, incremental behaviors, is to capture *when* behaviors change and *how* they change. Semantically we interpret them as a triple of an event (e), an initial value (v_0) and an accumulation function (f):

$$[\![IBehavior_{\tau,\delta}]\!] = \{(e, v_0, f) \in [\![Event_\delta]\!] \times [\![\tau]\!] \times ([\![\tau]\!] \times [\![\delta]\!] \to [\![\tau]\!])\}$$

An incremental behavior has two type parameters, τ denotes the behavior's value while δ is the type of its increments. The event component in the semantics refers to the increment responsible for the change in a behavior's value. The type signature of the `fold`† operation on events in fig. 1 is the motivation behind the semantics. From now on we replace `fold`† (which creates `Behaviors`) with `fold` to create incremental behaviors:

```
trait Event {
  ...
  def fold[B](init: B)(accum: (B, A) ⇒ B): IBehavior[B]
}
```

Generally, an incremental behavior is a behavior that has been, or could have been, defined using `fold`. In other words, it can be seen as a reified `fold`. To work with incremental behaviors we provide the following functions: `constant`, `incMap`, `incMap2`, `snapshot` and `toBehavior`, shown below:

```
trait OneOrBoth[+A, +B]

case class Left[A](l: A) extends OneOrBoth[A, Nothing]
case class Right[B](r: B) extends OneOrBoth[Nothing, B]
case class Both[A, B](l: A, r: B) extends OneOrBoth[A, B]

object IBehavior {
  def constant[A, DA](init: A): IBehavior[A, DA]
}
trait IBehavior[A, DA] {
  def deltas: Event[DA]
  def incMap[B, DB](valueFun: A ⇒ B)(deltaFun: (A, DA) ⇒ DB)
                   (accumulator: (B, DB) ⇒ B): IBehavior[B, DB]
  def incMap2[B, DB, C, DC](b: IBehavior[B, DB])(valueFun: (A, B) ⇒ C)
                           (deltaFun: (A, B, OneOrBoth[DA, DB]) ⇒ Option[DC])
```

```
                          (accumulator: (C, DC) ⇒ C): IBehavior[C, DC]
  def snapshot[B, C](ev: Event[B])(f: (A, B) ⇒ C): Event[C]
  def toBehavior: Behavior[A]
}
```

`constant` and `snapshot` work exactly as they do on regular behaviors by creating constants and allowing behaviors to be sampled at the rate of events. The chosen semantics for incremental behaviors make their semantic implementation trivial, but the complexity of folding events is moved to `toBehavior`.

It executes the fold and turns an incremental behavior into a continuous one ($Time \to \tau$):

$$toBehavior : IBehavior_{\tau,\delta} \to Behavior_\tau$$
$$[\![toBehavior]\!](\ (e, v_0, f)\ ) = \lambda t. f(f(...(f(v_0, d_1), ...), d_{n-1})d_n)$$
$$\textbf{if}\ (t_1 < ... < t_n \leq t < t_{n+1} < ...) \wedge \{(t_1, d_1), ..., (t_n, d_n), (t_{n+1}, d_{n+1})...\}\ = e$$

toBehavior defines a behavior that, upon evaluation at a time t, returns the accumulation (using f) of all event values up to time t, starting from the initial value (v_0).[4] Note that while `toBehavior` is an explicit method in this paper, a subclass relation between incremental behaviors and behaviors is completely reasonable.

`incMap` has the same purpose as a behavior's `map`, that is, provide a way to apply a function over the data. In the case of incremental behaviors we require three things. (1) f is the function that maps the old value of an incremental behavior to the new. (2) f_δ maps old deltas to new deltas, and (3) `accumulator` tells us how to put new values and new deltas back together. Note that we expect the programmer to take care of a proper relation between the old accumulator acc_{old}, f_δ, f and the new accumulator acc_{new}:

$$f(acc_{old}(\alpha, \delta_\alpha)) = acc_{new}(f(\alpha), f_\delta(\delta_\alpha)) \qquad \forall \alpha \in A. \forall \delta_\alpha \in DA$$

`incMap2` is also more complex than a behavior's `map2`. But its purpose is also the same, that is, provide a way to combine two behaviors into one. Its main parameters are a second behavior and a combination function, but two additional parameters are required to produce an incremental behavior. The first, `deltaFun`, takes two values of the incoming behaviors as well as a value of type `OneOrBoth[DA, DB]`. This type contains either an increment of the first or the second behavior (of type `DA` resp. `DB`) or both. `deltaFun`'s task is to compute an increment of type `DC` that represents the change (if any) that the given changes cause in the value of the resulting behavior. The final parameter `accumulator` tells us how to apply the new type of increments to previous values, it is the fold function for the new incremental behavior.

Fixing Todo List. We fix the overhead issues that were present in the previous example from fig. 2 by using incremental behaviors in fig. 3. We omit the creation of `submissionE` since it is identical to the implementation in fig. 2.

[4] This construction assumes that the event fires only a finite amount of times before any fixed time t (a property we call uniform discreteness).

```
val todoListIB: IBehavior[List[Entry], Entry] =
  submissionE.fold(List.empty) { (list, entry) ⇒ entry :: list }
val todoListViewIB: IBehavior[String, String] =
  todoListIB.incMap { _.map(_.pretty).mkString } { _.pretty }
                    { (accStr, dStr) ⇒ dStr + "\n" + accStr }
def replicate(ib: IBehavior[String, String]) = ...
replicate(todoListViewIB)
```

Fig. 3: FRP Incremental Todo List Example

We create an incremental todo list with incremental state by using `fold` to create `todoListIB`. In this example we create an incremental version of the pretty printed todo list (`todoListViewIB`) by using `incMap`, the incremental version of `map`. It takes three arguments. The first defines how to create a pretty printed string from a list of entries by mapping it: `_.map(_.pretty).mkString`. The second defines how the deltas should change by pretty printing the old delta: `_.pretty`. The final argument tells us how to combine our new values with the new deltas through string concatenation: `(accStr, dStr) ⇒ dStr + "\n" + accStr`.

The result is a version of the pretty printed todo list that is synchronized with the actual state through a time complexity of $\mathcal{O}(1)$ instead of $\mathcal{O}(n)$.[5] Similarly, `replicate` can now be implemented efficiently since it has access to a behavior's fine-grained change. It can directly send just a trace of its changes.

3 Incremental Behaviors as a Foundation

The advantages of incremental behaviors are apparent with performance improvements and the ability to model with behaviors where they are appropriate, such as using a behavior for the string representation of our todo list. However, they came at a cost, the API of incremental behaviors is more complex than their non-incremental counterparts. While the general incremental behavior API offers the most freedom, its complexity can be off-putting.

Other work on different FRP primitives such as incremental collections [14, 17] and discrete behaviors [15, 20, 3] provide similar benefits in certain cases while having a much simpler API. After our general proposal we now demonstrate that these other approaches can be seen as specialized versions of incremental behaviors by implementing them on top of our design.

3.1 Incremental Collections

Compared to regular behaviors, it is harder to create composable incremental behaviors. Let us use `todoListIB` from fig. 3 as an example. We create a function `toView` that takes an incremental behavior of entries and returns a view such as `todoListViewIB`:

[5] To keep the code concise we ignore the inefficient string operations here.

```
trait RSeq[A] {
  def map[B](f: A ⇒ B): RSeq[B]
  def filter(f: A ⇒ Boolean): RSeq[A]
  def foldUndo[B](init: B)(op: (B, A) ⇒ B)(undo: (B, A) ⇒ B): Behavior[B]
  def flatMap[B](f: A ⇒ RSeq[B]): RSeq[B]
}
```

Fig. 4: Reactive Sequence Core API

```
def toView(ib: IBehavior[String, String]) =
  ib.incMap { _.map(_.pretty).mkString } { _.pretty }
            { (accStr, dStr) ⇒ dStr + "\n" + accStr }
val todoListIBF: IBehavior[List[Entry], Entry] =
  submissionE.fold(List.empty) { (list, e) ⇒ if (e.title.contains("FRP")) e :: list
                                             else list }
```

Using `toView` on `todoListIB` creates an incremental behavior identical to the previously defined `todoListViewIB`. We define a second version of `todoListIB` that filters out entries that do not contain FRP within their title (`todoListIBF`). Using `toView` on `todoListIBF` instead does not create a properly pretty-printed filtered to-do list version. The problem here is that, although `todoListIBF` also uses `Entry` as the type of deltas, `toView` fails to take into account the different meaning of the delta type: a delta of type `Entry` is unconditionally added to the list in `todoListIB`, but only under a certain condition in `todoListIBF`. This example illustrates a general issue with incremental behaviors: functions operating on them are now not just coupled to the representation of data but also to the representation of deltas.

However, for standard types with standard APIs (like collections), we can mitigate this problem by defining a standard type of deltas (with a standard meaning). Both [14] and [17] propose incremental collections in a reactive programming environment to get more efficient collection operations without adding the API complexity that incremental behaviors bring. For this section, we focus on [14]'s abstraction: an incremental sequence (`RSeq[A]`) and discuss how it can be implemented on top of incremental behaviors. From a high level you can think of it as an efficient version of `Behavior[Seq[A]]`. Its usage and API is similar to collection libraries as shown in fig. 4.

A commonly used operation is `map`, which for reactive sequences returns a new reactive sequence. The mapped `RSeq` does not remap the entire list upon change, instead modifications to the list are processed on their own. Elements that should be inserted are mapped separately and their results are inserted directly. The same goes for deletions, which are propagated to the mapped list and directly remove an element. Other common collection operations work similarly.

We implement reactive sequences as a special kind of incremental behavior: `RSeq[A]` $\simeq$ `IBehavior[Vector[A], SeqDelta[A]]`. Our Scala prototype implementation of incremental behaviors also contains a collection library. It implements incremental collections by using a standard delta that models common operations such as addition or deletion. Incremental collection APIs are imple-

```
type IVector[A] = IBehavior[Vector[A], SeqDelta[A]]

sealed trait SeqDelta[+A] {
  def apply(v: Vector[A]): Vector[A]
}
case class Insert[A](element: A, index: Int)              extends SeqDelta[A]
case class Remove[A](element: A)                          extends SeqDelta[A]
case class Update[A](element: A, index: Int)              extends SeqDelta[A]
case class Combined[A](d1: SeqDelta[A], d2: SeqDelta[A])  extends SeqDelta[A]

def updated[A](iv: IVector[A], updates: Event[(A, Int)]): IVector[A]
def insert[A](iv: IVector[A], insertions: Event[(A, Int)]): IVector[A]
def remove[A](iv: IVector[A], deletions: Event[Int]): IVector[A]
```

Fig. 5: Reactive Sequence Implementation

```
def mapDelta[A, B](d: SeqDelta[A], f: A ⇒ B): SeqDelta[B] =
  d match {
    case Insert(element, i)    ⇒ Insert(f(element), i)
    case Remove(element)       ⇒ Remove(f(element))
    case Update(element, i)    ⇒ Update(f(element), i)
    case Combined(d1, d2)      ⇒ Combined(mapDelta(d1, f), mapDelta(d2, f))
  }
def map[A, B](rseq: IVector[A])(f: A ⇒ B): IVector[B] =
  rseq.incMap(v ⇒ v.map(f))(d ⇒ mapDelta(d, f)) {
    (acc: B, delta: SeqDelta[A]) ⇒ delta.apply(acc)
  }
```

Fig. 6: Map Implementation

mented through incremental behavior operations such as `incMap2`. It plugs into the Scala standard library and uses the appropriate collection abstractions such as traversable and sequence to provide a generic incremental API for the collection library.

In fig. 5, we demonstrate the model of such an incremental collection. Do keep in mind that to avoid Scala-specific concepts we focus on a reactive sequence implementation for just the vector and that an implementation for generic traversables or sequences is more complex.

In short, the incremental vector that we build is a vector data structure that efficiently handles incremental changes based on incremental behaviors. As a first step we model the different types of incremental changes (`SeqDelta`s). Each increment contains a method `apply` that defines the application of the increment to a vector, for brevity we assume its implementation. The different types of increments that we support are: an insertion (`Insert`), a removal (`Remove`), an in-place update (`Update`) or a combination of other deltas (`Combined`). Insertions simply contain the element to be inserted at a specific index, removals contain the element that should be removed and updates contain an element that should replace an element on a specific index.

They correspond to the three functions that are available on the incremental vector: `updated`, `insert` and `remove` as shown in fig. 5.

Using the model from fig. 5, we implement a simple version of an incremental map in fig. 6. Its created by using the incremental behavior's `incMap` function. The first argument provides a way to transform the initial vector to an initial result vector. In our case this is a simple map: `v ⇒ v.map(f)`. The second argument contains the transformation on the increments. For this we defined a helper function that applies the function `f` and use it accordingly:`d ⇒ mapDelta(d, f)`. The final argument defines how new deltas are applied to new values, in our case it remains the assumed `apply` method on `SeqDelta`.

For a discussion about higher order APIs on reactive sequences such as `flatMap` we refer to Section 5.

3.2 Discrete Behaviors

Continuous behaviors change unpredictably and continuously and their semantics are simple because their meaning are functions of time ($Time \to \tau$). But, in practice, discrete behaviors are often used [15, 20, 3]. The semantics provide less freedom but they express discrete changes. Essentially, they capture and expose *when* behaviors change. They are represented as an initial value and a stream of value changes:

$$[\![DBehavior_\tau]\!] = \{(e, v_0) \in [\![Event_\tau]\!] \times [\![\tau]\!]\}$$

Other than exposing the time at which they change (`def changes: Event[A]`), their API is identical to that of continuous behaviors.

It turns out that discrete behaviors can be implemented as a special case of incremental behaviors (`DBehavior[A]` $\simeq$ `IBehavior[A, A]`) with a trivial implementation for the accumulator. Their simple behavior API is possible since the accumulator never changes and both types are the same, for example an implementation of `map`:

```
type DBehavior[A] = IBehavior[A, A]
def accumulator[A](oldV: A, newV: A) = newV
def map[A, B](b: DBehavior[A])(f: A ⇒ B): DBehavior[B] = b.incMap(f)(f)(accumulator)
```

Incremental Behaviors through Computing Differences. There are situations where bandwidth is costly such as saving a behavior's history to disk (for logging or debugging) or sending its data across the network. When time complexity is of less concern, the API complexity of incremental methods (such as `incMap2`) makes them less desirable, regardless of their computational benefits.

To accommodate these scenarios we propose a way to obtain incremental behaviors through the simpler discrete behaviors:

```
trait DBehavior[A] {
  def toIBehavior[DA](diff: (A, A) ⇒ DA)(patch: (A, DA) ⇒ A): IBehavior[A, DA]
  def toIBehaviorGeneric[DA](implicit d: Delta[A, DA]): IBehavior[A, DA]
}
```

We recover increments between values by computing their differences. We require `diff` and `patch` functions to be defined with the following relation:

$patch(v_1, diff(v_2, v_1)) = v_2 \quad \forall v_1, v_2 \in A$. `diff` is used to compute differences that work as deltas and `patch` completes the incremental behavior by defining how they fold back into a value. In practice these two functions can often be derived with generic programming approaches. For example, there is a Scala library[6] which could be integrated; in this case the user would see an API like `DBehavior.toIBehaviorGeneric`. Using it is easy, for example, converting `rates`:

```
val rates: DBehavior[Set[Rate]] = ...
rates.toIBehaviorGeneric
```

4 Evaluation

Accompanying our proposed incremental behavior API is a prototype implementation of our ideas. We evaluate this prototype through some microbenchmarks to confirm our expected results. Note that neither our FRP implementation nor our incremental collection implementation were built with performance in mind and as such, overhead is not as low as it could be.[7]

Computational Performance. In fig. 7 we demonstrate the results regarding the computational performance of incremental behaviors. This microbenchmark is similar to the todo list example of fig. 3. We start with an initial vector of a certain amount of integers (plotted on the x-axis) and map them with a function that generates 10 random numbers. Afterwards, we update a single value in the collection and propagate the change to the mapped collection (time to update plotted on the y-axis). As a base case we use a regular push-based (discrete) behavior that contains a Scala vector from the collection library (marked `DBehavior[Vector[Int]]`). We compare the base case to three different incremental behaviors. First, a hand coded implementation using the `fold` primitive, exactly as we did in the todo list example from fig. 3 (marked `IBehavior[Vector[Int], Int]`). Second, an implementation based on the incremental collection abstraction from Section 3.1 (marked `ICollection[Int, Vector[Int]]`). Finally, an implementation which starts from the base case vector and performs a naive diffing comparing all elements of the vectors in order to convert a discrete behavior into an incremental one as discussed in section 3.2 (marked `DBehavior[Vector[Int]]` $\Rightarrow$ `ICollection[Int, Vector[Int]]`).

The graphs show that, as expected, the naive approach of the regular behavior is the slowest. It remaps every element in the vector whenever a change is made since it does not propagate what changed. The hand coded example has the best performance, it is hard-coded to handle only one case, modifying the head of the collection. It maps the element in isolation and then creates a new modified Scala vector. Similarly, the incremental collections vector also isolates changes,

[6] `https://github.com/stacycurl/delta`

[7] To make our benchmarks as accurate and fair as possible on the JVM, the measurements are results over multiple VM invocations, each prepared with warmup operations.

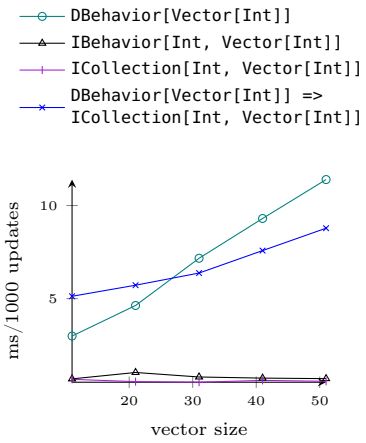

Fig. 7: Updating mapped vectors

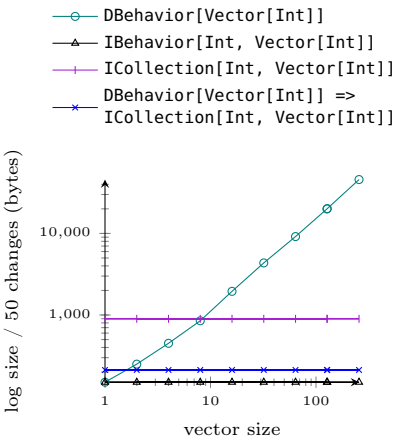

Fig. 8: Updating logged vectors

however it has to go through an added abstraction layer that handles other cases than just changing the head of a collection, this causes a bit of overhead. Finally, the case of recomputing the differences. Recomputing the differences trades remapping for a difference algorithm, depending on the algorithm the outcome may be worth it.

Bandwidth Overhead. In fig. 8 we demonstrate the results of a scenario where bandwidth matters. The goal of the application is to write a trace to disk from which all versions of values can be recomputed. This mimics scenarios where bandwidth is important, such as replicating a behavior's value across the internet. Its implementation is similar to the todo list example in fig. 3. We change one element in the collection several times and after every change we expect to be able to trace a trail of its value on disk. The most efficient implementation logs the first value in its entirety and adds differences afterwards.

We test the same four cases as the last benchmark. We can immediately see that no matter the vector's size, the incremental implementations remains constant in storage size since their differences remain identical (1 value change). Again, there is an overhead for the higher abstractions used. Extra information regarding the change to the collection is encoded by the incremental collections vector which implies a larger bandwidth footprint. The hand coded result only supports the specific case of the benchmark and logs the absolute minimum at the cost of added programmer complexity. The base case is a naive implementation that logs the entire vector every time.

5 Discussion: Higher-Order Incremental Behaviors

In the paper we used *first-order FRP* instead of higher order FRP. Concretely, this means that we do not provide APIs that are higher-order in the sense that they work with events of events or behaviors of behaviors. A typical example is:

```
def join[A](b: Event[Event[A]]): Event[A]
```

It takes an event that fires new events and returns an event that dynamically includes other events.

Higher-order APIs is a tricky subject because the natural semantics for APIs like `join` cannot be implemented without storing or being able to recalculate all previous values of any generated behavior or event that is incorporated into the network. This implies a very high memory usage, which is known in the FRP literature as *time leaks*, see [22] for more details. Various solutions have been proposed for avoiding time leaks, often based on restricting which `Events` may fire from an argument of APIs like `join`. One particular solution which we like to highlight is [13], an approach that eliminates such leaks by introducing a new kind of type-based capability that grants the right to allocate memory.

We do not go into more details on higher-order FRP or the ways to avoid time leaks, because we believe incremental behaviors are orthogonal to the problem. If we assume the existence of a naive higher-order `flatMap` API for standard events (as used by [7]), then we can implement a higher-order primitive for incremental behaviors. Concretely, we assume a higher-order API with the following function that lets us flatten nested events:

```
def flatMap[A, B](e: Event[A], f: A ⇒ Event[B]): Event[B]
```

In this setting, we can implement a higher-order operation for incremental behaviors that looks as follows:

```
trait IBehavior[A, DA] {
  private val initial: A
  def incFlatMap[DB, B](fa: A ⇒ IBehavior[B, DB])
                       (fb: (A, DA) ⇒ Event[DB])
                       (accumulator: (B, DB) ⇒ B): IBehavior[B, DB] = {
    val newDeltas: Event[DB] = this.deltas.flatMap(fb)
    val newInitial: B        = fa(this.initial).initial
    newDeltas.fold(newInitial)(accumulator)
  }
}
```

Similarly to `incMap`, the first argument maps the old values to the new, while the last argument redefines the accumulator. But, the second argument allows new increment events to be inserted (`DA ⇒ Event[DB]`). In other words, it enables dynamic insertion of deltas into the incremental behavior. Use of `incFlatMap` is only correct if `fa`'s behavior and `fb`'s event have the following relation: $\forall a.\, \forall da.\, fb(a, da) = fa(applyDelta(a, da)).changes$

Interestingly, this `incFlatMap` on incremental behaviors and `join` on events is all that is required to implement the missing higher order methods from reactive sequences in Section 3.1.

6 Related Work

We split our related work section into two categories: (1) functional reactive programming and its semantics, and (2) incremental computing, and more specifically self-adjusting computations (SAC).

Functional reactive programming was first introduced in FRAN [6] to write composable reactive animations. However, its proposed semantics are inherently leaky in the higher-order primitives [see 22, sec. 2]. Since then multiple revisions of the semantics were proposed [see among others, 10, 7, 13, 21, 22].

In contrast, our work focuses on extending the first-order semantics with support for explicit incremental computations. The incremental behaviors that we propose are a generalization of patterns that appear in other work like incremental lists [14, 17] or discrete behaviors [15, 20, 3]. An implementation of incremental behaviors that is, as far as we can tell, close to the semantics proposed in this paper can be found in the grapefruit library [12], but its semantics are not written down in documentation or an accompanying paper.

Incremental Computation is a way of implementing programs that do not redo entire calculations after a change in input. We divide this work based on how much manual work a programmer has to do.

Some approaches based on memoization [18] and self-adjusting computations (SAC) [1] minimize manual interference by using dependency graphs and propagation algorithms to efficiently react to input changes. In practice, this has several similarities with FRP, which is also frequently implemented by propagating changes through dependency graphs. However, their focus and granularity differ. SAC focuses on efficiently reacting to small changes in input while FRP focuses on providing simple denotational semantics for event-driven programs. Our FRP work with incremental behaviors is a middle ground which allows programmers to manually express a finer granularity compared to traditional FRP. It allows incremental computations that use the FRP implementation's propagation and dependency tracking to be defined by the user. [4] describes a different approach to automated incremental computations. They define ILC, a static and extendable program-to-program transformation that lifts incremental computations on first-order programs to incremental computations on higher-order programs.

A different approach to automated solutions are frameworks that help programmers with writing incremental computations. Instead of trying to automate the encoding of incremental algorithms it aims to make such computations easier to write such as the reactive sequences [14]. We discuss two other examples in detail:

i3QL [16] proposes relational algebra as a suitable API for incremental computing. i3QL operators are a high-level abstraction for incremental computing implemented on top of a low-level Observable-like change-propagation framework that supports events for adding, removing and updating values. Incremental behaviors are an addition to traditional FRP to further capture when and how

values change, it's more akin to the lower-level implementation of i3QL's propagation framework (which supports 'when' and 'how' natively). It would be very interesting to see to which extent incremental behaviors and FRP can be used to implement i3QL's relational algebra operators.

[8] describes a general framework in which incremental changes and their propagation are made to compose through typeclasses. They define a type class for changes:

```
class Change a where
  type Value p :: *
  ($$) :: p -> Value p -> Value p
```

A change of some type has a value to which it can be applied using the `$$` function. Incremental operations can be defined as transformation from one type to another. A transformation in its simplest form is a pair of two functions:

```
data Trans p q = Trans (Value p -> Value q) (p -> q)
```

The similarity between these transformations and our `incMap` on behaviors is interesting. `incMap` takes three functions as arguments, one turns a state `A` into another, a change `DA` into another and finally a function that defines how the new change and the new state can be combined. This corresponds exactly to the `Trans` and `Change` functionality. `Trans` defines how to convert both values and changes from one type to another while `Change` makes sure that there is a way to apply changes to a value.

Note that they propose several more complicated versions of `Trans`, e.g.:

```
data Trans p q = forall s. Trans (Value p -> (Value q, s)) (p -> s -> (q, s))
data Trans p q = Trans (forall s. Value p -> ST s (Value q, p -> ST s q))
```

The first provides access to a local pure state. Something that can be emulated with `incMap`. The second version provides *safe* access to local mutable state, something for which we do not have an alternative. Given the similarites between the two approaches a logical step for future work seems to incorporate the typeclass-based framework for incremental computing into the incremental behavior API which could give us alternative definitions such as:

```
trait ITBehavior[A: Change] {
  def incMap[B: Change](trans: Trans[A, B]): ITBehavior[B]
}
```

As a final note, Several SAC papers [2, 9] refer to FRP and point out opportunities to combine the two, for example, [2] says "Although FRP research remained largely orthogonal to incremental computation, it may benefit from incremental computation, because computations performed at consecutive time steps can be similar.". Incremental behaviors may be a stepping stone towards a better integration of incremental computations in FRP.

7 Conclusion & Future Work

Functional Reactive Programming in theory is nice and simple, but in practice there are some shortcomings that prevent general use. This paper tackles one

such problem, the ability to efficiently deal with large incrementally constructed values.

We presented incremental behaviors, a new FRP primitive that can express *how* and *when* values change. We show that it can be used as a foundation for other abstractions and demonstrate that its benefits are noticeable by executing a performance analysis and comparing it to traditional FRP.

As future work, applying ideas from self-adjusting computations to further automate the writing of incremental programs sounds promising. An automatic and efficient replacement to our `diff` function which converts discrete to incremental behaviors efficiently would be the ideal case.

Acknowledgments. Bob Reynders holds an SB fellowship of the Research Foundation - Flanders (FWO). Dominique Devriese holds a postdoctoral fellowship of the Research Foundation - Flanders (FWO).

Bibliography

[1] Acar, U.A., Blelloch, G.E., Harper, R.: Adaptive functional programming. TOPLAS 28(6), 990–1034 (2006)

[2] Acar, U.A., Blume, M., Donham, J.: A Consistent Semantics of Self-Adjusting Computation. J. Funct. Program. 23(03), 249–292 (2013)

[3] Blackheath, S.: Denotational semantics for Sodium. `http://blog.reactiveprogramming.org/?p=236` (2015)

[4] Cai, Y., Giarrusso, P.G., Rendel, T., Ostermann, K.: A theory of changes for higher-order languages: Incrementalizing λ-calculi by static differentiation. In: PLDI. vol. 49, pp. 145–155. ACM

[5] Czaplicki, E., Chong, S.: Asynchronous Functional Reactive Programming for GUIs. In: PLDI. pp. 411–422. ACM (2013)

[6] Elliott, C., Hudak, P.: Functional Reactive Animation. In: ICFP. pp. 263–273. ACM (1997)

[7] Elliott, C.M.: Push-pull Functional Reactive Programming. In: Haskell. pp. 25–36. ACM (2009)

[8] Firsov, D., Jeltsch, W.: Purely Functional Incremental Computing. In: Brazilian Symposium on Programming Languages. pp. 62–77. Springer (2016)

[9] Hammer, M.A., Dunfield, J., Headley, K., Labich, N., Foster, J.S., Hicks, M., Van Horn, D.: Incremental computation with names. In: OOPSLA. vol. 50, pp. 748–766. ACM (2015)

[10] Jeltsch, W.: Signals, Not Generators! Trends Funct. Program. 10, 145–160 (2009)

[11] Jeltsch, W.: Strongly Typed and Efficient Functional Reactive Programming. Ph.D. thesis, Universitätsbibliothek (2011)

[12] Jeltsch, W.: Grapefruit-frp: Functional Reactive Programming Core. `https://hackage.haskell.org/package/grapefruit-frp` (2012)

[13] Krishnaswami, N.R.: Higher-order functional reactive programming without spacetime leaks. In: ICFP. vol. 48, pp. 221–232. ACM, `http://dl.acm.org/citation.cfm?id=2500588`

[14] Maier, I., Odersky, M.: Higher-Order Reactive Programming with Incremental Lists. In: ECOOP. pp. 707–731. Springer-Verlag (2013)

[15] Maier, I., Rompf, T., Odersky, M.: Deprecating the observer pattern. Tech. rep. (2010)

[16] Mitschke, R., Erdweg, S., Köhler, M., Mezini, M., Salvaneschi, G.: i3ql: language-integrated live data views. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications. pp. 417–432. ACM

[17] Prokopec, A., Haller, P., Odersky, M.: Containers and Aggregates, Mutators and Isolates for Reactive Programming. In: SCALA. pp. 51–61. ACM (2014)

[18] Pugh, W., Teitelbaum, T.: Incremental Computation via Function Caching. In: POPL. pp. 315–328. ACM (1989)

[19] Reynders, B., Devriese, D., Piessens, F.: Multi-Tier Functional Reactive Programming for the Web. In: Onward! pp. 55–68. ACM (2014)
[20] Salvaneschi, G., Hintz, G., Mezini, M.: REScala: Bridging between object-oriented and functional style in reactive applications. In: MODULARITY. pp. 25–36. ACM (2014)
[21] van der Ploeg, A.: Monadic Functional Reactive Programming. Haskell 48(12), 117–128 (2014)
[22] van der Ploeg, A., Claessen, K.: Practical Principled FRP: Forget the Past, Change the Future, FRPNow! In: ICFP. pp. 302–314. ACM (2015)
[23] Wan, Z., Taha, W., Hudak, P.: Real-time FRP. In: ICFP. pp. 146–156. ACM (2001)