Trampoline Variables: A General Method for State Accumulation in Reactive Programming

Bjarno Oeyen
bjarno.oeyen@vub.be
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium

Sam Van den Vonder
sam.van.den.vonder@vub.be
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium

Wolfgang De Meuter
wolfgang.de.meuter@vub.be
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium

Abstract

Reactive programming is all about relegating the management of a program's state changes to the realm of the runtime environment. Nevertheless, sometimes it is still necessary to enrich a reactive program with state variables that are explicitly updated by the programmer. In current reactive languages this is accomplished either by polluting the reactive paradigm with imperative constructs or by relying on built-in operators such as `foldp`.

This paper introduces *trampoline variables*, a new *general* mechanism that allows reactive programs to manipulate state explicitly without resorting to imperative programming. We show that our proposal is at least as powerful as existing built-in reactive operators. We also analyse how reactive programs with trampoline variables can be composed and how they can form the basis to replace stateful constituents of a running reactive program — a.k.a. hotswapping — in a coherent way. The latter is an essential building block towards live IDEs for reactive programming languages.

CCS Concepts: • **Software and its engineering → Data flow languages**.

Keywords: Reactive Programming, Higher-Order Programming, Stateful Computations

ACM Reference Format:
Bjarno Oeyen, Sam Van den Vonder, and Wolfgang De Meuter. 2018. Trampoline Variables: A General Method for State Accumulation in Reactive Programming. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS '21), October 18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Reactive Programming (RP) can be considered the exact opposite of imperative programming. The latter embraces the explicit management of state changes. The former hides it as much as possible. This raises the question on how to combine both paradigms, something which is typically needed whenever RP is used in an effectful or stateful context. We discern two problems:

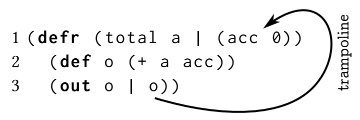
- Embedding RP code in an imperative world. This is largely understood. It corresponds to designing a "foreign function interface" in which the imperative program is given access to variables from the RP program [24].
- Embedding imperative code in an RP world. This is far less understood [17] even though the need for it is widely documented. E.g., a reactive program that computes the average of incoming numbers or a reactive program that is put in a certain state depending on the order of arriving values (i.e. a finite state automaton).

To implement a stateful computation in a reactive program, a programmer could rely on the imperative functionality provided by a "host language". E.g., in REScala [38], a reactive program is constructed by *lifting* ordinary Scala functions. By allowing said functions to perform side effects to standard Scala variables, the reactive program can manage state explicitly. The REScala manual discourages doing this, but this is not enforced by the language [1]. In [17], we demonstrated that this leads to reactive programs with ill-defined semantics.

Alternatively, an RP language can forbid imperative assignments. Instead, a language may provide a set of built-in operators that manage hidden state. E.g., in Elm [16], the operator `foldp` manages an implicit accumulation variable that is automatically updated by the RP runtime every time an input signal produces a new value. Even though the foundations of higher-order operators such as `fold` are well-understood in functional programming (see e.g.. [23]), the implementation of `foldp` is hardwired into the RP language and there exists no *general mechanism* that allows a programmer to build their own reusable stateful reactive operators. Furthermore, these operators completely hide the stateful variables in their implementation, which, especially in the presence of hot-code reloading, makes it difficult to migrate state between different code versions.

In this paper, we present *trampoline variables*: a *general* mechanism, for maintaining state in RP programs that does not confront programmers with the problems resulting from unrestricted assignment statements in their reactive code, while avoiding a built-in set of stateful operators (like `foldp`). We present our ideas in an RP language called Haai (pronounced "high"). In Haai [33], a reactive program corresponds to a so-called *reactor*. A reactor can be seen as small

REBLS '21, October 18, 2021, Chicago, IL, USA
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn



```
(defr (total a | (acc 0))
  (def o (+ a acc))
  (out o | o))
```

Listing 1. One source, one sink and one trampoline variable.

```
1 val numbers: Event[Double] =
2   ??? /* definition not important for the example */
3 var state: (Double, Int) = (0d, 0)
4 numbers.map(n => state = (state._1 + n, state._2 + 1))
5 val average = Signal { state._1 / state._2 }
```

Listing 2. Updating `state` does not recompute `average`.

(piece of a) reactive program that is represented by a DAG: it has input nodes, internal nodes and output nodes. Reactors can be composed within one another and deployed (i.e. instantiated) on time-varying signals. Trampoline variables will become "implicit" input variables of a reactor. Inside the reactor, they can be used like any other variable. However, at the very end of the reactor code (together with the output nodes of the reactor) additional expressions specify how the trampoline variables must be updated. The next time the reactor needs to react, its trampoline variables will be bound to the updated values. Listing 1 gives a sneak preview of the mechanism. The reactor `total` will react to some input signal `a`. It defines one trampoline variable `acc` (short for accumulator) that is initialised to 0. Whenever `a` produces a new value, the signal `o` is updated (by performing the addition of `a` and `acc`) and its value will be the output produced by the `total` reactor. Hence the reactor has one input node and one output node. However, after the `|` symbol, the expression `o` is used to specify the value of `acc` to be used in the next turn when `a` produces a value. With this behaviour, the `total` reactor reactively computes a running total of a given input signal.

In this paper, we show that:

- Trampoline variables are a general mechanism that can be used to implement the stateful operators that exist in several other RP languages.
- Trampoline variables allow us to compose first-class stateful reactors in an elegant way. Trampoline variables interact with nested reactors in a clean way.
- Reactors with trampoline variables are the exact level of granularity that is needed to replace parts of a stateful reactor program while it is reacting, i.e. hot-code reloading.

The paper is structured as follows. In Section 2 we give an overview of the stateful operators that exist in various kinds of existing RP languages. In Section 3 we present a brief overview of Haai's semantics, and Section 4 presents the extension of Haai that provides trampoline variables. We focus on their core semantics and how they interact with Haai's existing semantics. In Section 5 we show that trampoline variables cover most existing stateful operators. Finally, in Section 6 we show how trampoline variables form the basis of hot-swapping stateful reactive programs in running Haai programs, by retaining trampoline variables during a hot-code reload. This is an essential ingredient for scalable live programming environment for RP.

2 Stateful Reactive Programming

Most formulations of RP (such as FrTime [13], Flapjax [29] and REScala [38]) are based on *function lifting*. The lifted function is typically written in the host language being extended with reactive concepts. If the host language is impure, the lifted function can refer to and update variables that are in its lexical scope. This model of managing state breaks down in at least 2 ways:

- A compiler of an RP language can update the nodes of the program as long as the update order respects the DAG's dependencies. Since the aforementioned lexical variables are not a part of the DAG, this can inadvertently reorganise assignment statements leading to ill-defined update semantics. In [17] this phenomenon was called **The Reactive Update Order Leak**.
- If a variable occurs in the lexical scope of 2 lifted functions and one of them updates the variable, this will not trigger a reactive re-evaluation of the other one. After all, the compiler cannot track the dependency since the body of the lifted functions falls outside the control of the reactive language. Listing 2 exemplifies this problem in REScala: the variable `state` is updated by the signal on Line 4, and referred to by `average`. Since `average`'s reference to `state` is not a reactive dependency, it is never re-evaluated. We call this **The Reactive Lexical Scope Leak Problem**.

A more disciplined approach consists of disallowing (or discouraging) destructive updates to lexically scoped variables or even to disallow impure lifted functions altogether. Languages can instead provide built-in stateful operators. For example, in Elm [16] the higher-order operator `foldp` creates a stateful reactive computation. Given a (pure) binary operator, an initial value, and a signal, `foldp` returns a new signal whose value consists of the accumulating state that is produced by applying the binary operator to the previous state and the value produced by the input signal, every time the input signal changes.

Many other RP languages feature similar stateful operators. Table 1 presents an overview of the stateful operators that we found in the literature. Note that different languages often use different names for the same (or very similar) operators. For simplicity, we have decided to group these together under a single name. For example, FrTime [13] does not have an operator named `foldp`, but it has two other operators (`collect-b` and `accum-b`) which perform the same kinds of accumulation, and in REScala, `foldp` is simply known as

Table 1. Overview of stateful operators found in various RP languages.

Name	Description	Languages
Domain Specific Stateful Operators		
`integral`	Given a signal carrying numbers, computes its Riemann integral with respect to time. A similar domain specific stateful operator is `derivative` which is found in FrTime [13].	AFRP [31], Fran [19], Frappé [15], and FrTime [13].
First-Order Stateful Operators		
`pre`	Given a signal s_0 and a initial value v, creates a signal s whose value is initially equal to v. Each time the signal s_0 produces a value, the previous value of s_0 is produced on s and the current value is stored, internally, to be used the next time s_0 produces a value.	ActiveSheets [46], Hae [50], and RT-FRP [48].
`last`	Similar to `pre`, except that no initial value needs to be passed to `last`. Instead, signals (on which `last` is being applied to) are annotated with an initial value in their definition, such that `last` can always provide an initial value.	Emfrp [40], and XFRP [41].
`latch`	Given two signals s_0 and s_1, creates a signal s whose value is updated each time s_1 produces a value, the new value of s is equal to the value that s_0 had in the previous update of s_1 (or, if undefined like in the first turn, the current value of s_1).	ActiveSheets [46]
`hold`	In RP languages that make a distinction between continuous signals and discrete event streams, `hold` is used to convert the latter into the former. Given a discrete event stream s_0 and an initial value v, creates a new (continuous) signal whose initial value is equal to v, each time the discrete event stream produces a new value, that value is held in the signal s.	FrTime [13], Nettle [47], REScala [38], and Sodium [9].
`delay-by`	Given a signal s_0 and a time duration t (e.g., in milliseconds), creates a new signal s whose value lags behind approximately t time units w.r.t. to s_0 (with respect to some global clock).	Flapjax [29], and FrTime [13].
Higher-Order Stateful Operators		
`foldp`	Given an update function f, an initial value v and a signal s_0, creates a new signal s whose value is initially equal to v. When signal s_0 produces a value e, s's new value is computed by applying f on s's current value and e.	CFRP [43], Elm [16], Frenetic [20], Flask [28], FRPNow [45], FrTime [13], Gavial [37], Hae [50], Hokko [36], Midair [30], ReactiFi [42], REScala [38], and SFRP [10].
`feedback`	Given a signal function (i.e., a reactive program) of type SF (a, c) (b, c) (where (a, c) is the input type and (b, c) the output type) and an initial value v, creates a new signal function of type SF a b which contains the original signal function, connecting the output of type c to the input of type c with an implicit one-turn delay (i.e. like `pre`) in-between, initialised with v.	Dunai [35], Hailstorm [39], Rhine [7], and Yampa [22].

`fold`. We have placed the operators into three disjoint categories. Domain-specific operators offer a specific functionality such as calculating the Riemann integral of a signal's values. The other operators are generic operators. First-order operators are deployed on a signal and always perform the same functionality. Higher-order operators make use of an additional function such as `foldp`'s binary operator.

While Table 1 features quite an extensive list of operators, an RP language often does not need to provide every operator as a primitive. For example, an RP language can choose to only provide `foldp` as a built-in primitive. Other stateful operators (e.g., like `pre` and `feedback`) can then be implemented using `foldp` and vice-versa (assuming that the base language is expressive enough).

Stateful operators are also present in various languages and libraries related to RP. We list a few examples below:

- **Synchronous Programming Languages** Scade 6 [12] has an operator named `pre`. Lucid Synchrone [11] has an operator named `fby` ("followed by") which is similar to the definition of `pre` in Table 1.

```
(defr (average-and-distance x y)
  (def avg (/ (+ x y) 2))
  (def dis (sqrt (+ (expt x 2) (expt y 2))))
  (out avg dis))
```

Listing 3. Reactor with two inputs and two outputs.

- **Streaming Frameworks and Libraries** Akka has an operator named `statefulMapConcat` [3] (which actually requires a programmer to make use of imperative assignments). RxJS [2] has operators like `count` and `withLatestFrom` which internally keep track of a counter or remember a past value of an observable.

A characteristic common to all these operators is that they are built into the RP (or streaming) language (or library) at hand. Unlike functional programming which is capable of explaining its built-in (higher-order) operators in terms of general mechanisms (e.g. first-class functions, recursion, tail-call optimisations), these stateful operators lack a general underpinning. Furthermore, these operators make it impossible to directly access state (as, by hiding the state in the implementation of the operator, it is impossible to name these stateful variables), which is necessary to retain state when hot-swapping reactive programs.

As a possible solution to these problems, we present "trampoline variables" which represent state that is directly encoded in a reactive program's dependency graph. We will describe their general usage and semantics in Section 4 and the advantages they have over ad-hoc operators (in the presence of hot-swapping) in Section 6. We first give a brief introduction to Haai, the experimental RP language that is used as our research vehicle.

3 Reactive Programming in Haai

We briefly describe Haai's syntax and core concepts to keep the paper self-contained. For a more detailed explanation on Haai, we refer to [32, 33].

3.1 Reactors, Signals and Deployments

Reactive programs in Haai are represented as **reactors** which are, in essence, directed acyclic dependency graphs with explicit source and sink nodes. The code in Listing 3 shows a reactor that, given two inputs x and y, produces their average as well as the distance to the origin (seen as two-dimensional coordinates).

3.1.1 Deploying a Reactor. Reactors themselves are static: they do not compute anything by themselves nor will they react to anything. In order for a reactor to come alive, it first needs to be "instantiated" (which we call *deploying a reactor*) by connecting its inputs to the right number of **signals** whose value (i.e. numbers, booleans[1], strings...) can change over time. The instances of reactors are called **deployments**. Each time when a reactor is deployed, new signals are created of which some will be used as output of the deployment (i.e. its sink signals). By default, the last expression is the reactor's only sink node. This behaviour is overriden when there is an `out` form present, as then the operands of the `out` form denote which the sink signals of the reactor are (as shown in Listing 3).

To bootstrap a Haai program, the Haai interpreter features a number of built-in signals such as `time`[2] as well as a number of built-in reactors (such as `+`, `/`, `sqrt` and `expt`). An important distinction w.r.t. many other RP languages is that Haai is a **pure RP language** that lacks functions. Reactors are the only building blocks to make programs, and even simple expressions like `(+ x y)` should therefore be thought of as the deployment of reactor `+` to signals `x` and `y` (and not of applying some built-in function to the values `x` and `y`). The deployment expression `(+ x y)` creates a new signal that depends on `x` and `y`. Each time either of those signals changes, that signal is updated accordingly.

The same reactor can be deployed several times and each deployment has its own set of sources, sinks and internal signals, and is thus completely independent from any other deployment of the same reactor. If one thinks of a reactor as a DAG that corresponds to the dependencies specified by the reactor, then – at least conceptually – every deployment corresponds to a copy of that DAG in which every node holds a current value. We therefore say that **reactors are re-entrant**. `average-and-distance` can thus be deployed multiple times, and changes to a signal in one deployment do not change the signals of another deployment of `average-and-distance`.

3.1.2 Reacting to Changes. Once a deployment has been made, any changes to any of the input signal causes an automatic re-computation by the RP runtime. Once a change has been pushed to one of the reactor's source signals, the RP runtime updates all signals in the deployment in topologically sorted order until its output nodes are reached. We will refer to this mechanism as *a turn*.

Turns occur whenever a primitive signal (like `time`) produces a value. Haai provides a number of built-in primitive operations (and commands) to connect the Haai interpreter to various external data sources, but a complete overview of this functionally is beyond the scope of this paper.

3.2 Higher-order Reactors

Reactors in Haai are able to produce reactor values. Consequently, this leads to the notion of higher-order reactors, which are reactors that can receive reactors on their input signals (and possibly deploy them) or produce reactor values as output. An example of a higher-order reactor that "takes" a reactor as input and "produces" one as output is shown

[1]Written as `#t` (true) and `#f` (false) like in Scheme [4].

[2]The built-in signal `time` corresponds to FrTime's `seconds`, except that it starts from 0 when the program starts instead of using Unix time.

```
(defr (add-or-mul r)
  (if (even? (r time))
      +
      *))
```

Listing 4. Reactor producing either + or *.

in Listing 4. add-or-mul[3] takes a signal of reactors r. Depending on whether whether the deployment of r on time results in an even number, add-or-mul's sink signal will produce the * or the / reactor. E.g., the deployment expression ((add-or-mull cube-or-square) a b) will produce $a+b$ or $a \cdot b$ depending on whether or not the values produced by (cube-or-square time) are even or odd.

Having higher-order reactors means that deployments can be dynamically "swapped into and out of" the reactive program's global DAG. Deployments of add-or-mul will continuously swap between different DAGs every time r produces a value. The implementation strategy to do this efficiently is also beyond the scope of the paper.

3.3 Anonymous Reactors & Captures

Analogous to anonymous functions, Haai features **anonymous reactors**. These are reactors that are defined in "the middle of" another reactor's definition. Just like anonymous functions, they are denoted by a Greek letter. Instead of lambda, like in functional programming, we denote anonymous reactors by the Greek letter rho (to emphasise the creation of a reactor).

Listing 5 shows an example of a reactor which internally contains an anonymous reactor definition. The reactor make-adder has one source (a) and contains one output expression which creates an anonymous reactor (which also has one source, b). When make-adder is deployed (e.g., on mouse-x), it produces a so-called **capture** which is similar to a closure in functional programming. At the interpreter level, a capture is a pair consisting of the rho's entire DAG and a reference to the deployment environment, i.e. the signals defined in its lexical environment. Hence, first-class reactors are represented as signals of captures. Built-in reactors such as + and * are, conceptually, captures with an empty deployment environment (i.e. they are global in the interpreter) and their corresponding signals are, at least conceptually, continuously producing these capture values.

When the capture from make-adder is deployed elsewhere in the program (e.g., on some other signal, say x-offset), a signal will be created (by the deployment of +) that depends both on mouse-x and x-offset. As such, the signals created by a capture deployment will not only produce a value when

[3]This example uses if, which creates a signal whose value is equal to that of either the consequent signal (second operand) or alternate signal (third operand), depending on the truth value of the condition signal (first operand). During each turn, the deployments needed to compute the value of the unused signal are disabled by if, thus (if #t 1 (/ time 0)) never produces a division-by-zero error.

```
(defr (make-adder a)
  (rho (b) (+ a b)))
```

Listing 5. Reactor using a lexically scoped signal.

any of their own sources (in this case x-offset) changes, but also if one of the signals from its lexical environment, which the capture makes use of, changes (in this case mouse-x). Signals in deployments of captures can thus change without any changes to their (explicit) sources. We call this **scope-driven reactivity**, as deployments of a capture also react to changes of signals in their lexical scope.

4 Trampoline Variables

Remember from Section 2 that our goal is to find a language mechanism that is generic enough to express all possible stateful reactive programs, without relying on a predefined set of built-in stateful reactors and without re-introducing imperative assignments.

4.1 Basic Idea

The main idea of the paper is to allow a reactor to declare a number of pseudo-inputs, called **trampoline variables** (trampolines for short). Trampoline variables are declared after the normal sources of a reactor, following a vertical bar (|). Each trampoline variable has of a name and an initial value (which are, syntactically, surrounded between parenthesis). We have already shown an implementation of a trampoline reactor (we will call reactors with a trampoline variable "trampoline reactors") earlier in Listing 1. We will now describe the semantic behaviour of total's trampoline variable. total has one trampoline variable named acc whose initial value is equal to 0 (the initial value will be used during the deployment of total to initialise the trampoline variable). At the end of the reactor definition, the vertical bar is used again in the out-form. The expressions that occur here are used to determine the trampoline variables' value for the next turn. Hence, there must be an equal number of expressions after the vertical bar in the out-form as there are trampoline variables. Every time the reactor reacts to one of its changing inputs, the trampoline variables will contain the values of these "trampoline out" signals of the previous turn.

The existence of trampoline variables in a reactor definition does not influence the input-arity nor the output-arity of a reactor. Thus, the total reactor has to be deployed on one input (source) signal and its deployment will have only one accessible output (sink) signal.

To exemplify the inner workings of total, Table 2 shows how each change on a signal a causes the deployment of (total a) to be updated. Each column shows how, in each turn t_i, the values of acc and o are updated with respect to the current value of a. A distinction is made in the table between the *current* and the *next* of the trampoline variable

Table 2. Evolution of the signals (and trampoline variable) inside `(total a)`, in function of `a`.

	t_1	t_2	t_3	t_4	…
a (input)	0	2	1	6	…
acc⇓	0	0	2	3	…
o (= a + acc⇓)	0	2	3	9	…
acc⇑ (= o)	0	2	3	9	…

`acc` (`acc⇓` and `acc⇑`, respectively). A fundamental property of trampoline variables is that they are *always* updated at the end of a turn. Thus, only when a turn is finished, is a trampoline variable updated with the contents of the corresponding signal defined in `out` part of the reactor definition. In other words, in every turn `acc⇓`'s value is equal to `acc⇑`'s value from the previous turn. Except in the first turn (t_1), when there is no previous value of `acc⇑`, where `acc⇓`'s value's is set to the trampoline's initial value, as specified in its specification.

In summary, trampoline variables declare state at the granularity of the reactor. Only the heading of the reactor (i.e. the declaration of the sources) and the `out`-part of the reactor (i.e. the declaration of the sinks) are aware of the existence of any trampoline variables. A reactor's internal expressions can pretend as if the trampoline variables are just ordinary (input) signals. Trampoline variables make it possible to remember values produced in a previous turn and reactive programs themselves do not react to changes of the trampoline variables, it is only when another signal in a deployment changes, that the value of the trampoline variable is used. As such, trampoline variables do not cause any cycles in the dependency graph of the reactive program.

4.2 Memory Footprint of Trampoline Variables

Trampoline variables are allocated and initialised when a reactor is deployed. Hence, every deployment of a reactor manages its own "copy" of that reactor's trampoline variables. The memory footprint of each reactor deployment is constant:

- Each trampoline variable stores one single value corresponding to the last value produced by a signal. At the end of each turn, the old value of a trampoline variable is discarded and replaced with the value of the signal as denoted in the `out`-part of the reactor. As such, trampolines in Haai do not cause any time-leaks [18].
- Furthermore, in the absence of data constructors (like pairs or arrays), no unbounded data structures can be created by trampoline variables. This ensures that Haai programs with trampolines can be executed in constant time with bounded memory usage, which is an important property of many reactive programs [32, 40, 48].

The introduction of state with trampoline variables is very similar to the introduction of state with tail-recursion

```
(defr (min-max s | (i s) (a s))
  (def mi (min s i))
  (def ma (max s a))
  (out mi ma | mi ma))
```

Listing 6. Simultaneously calculating min and max of `s`.

in process-oriented actor languages [26] such as Erlang [5] where an actor "updates" its state by calling a method in a tail-recursive way with new values for the parameters of that method. However, in order to avoid the thread hijacking problem, as explained in [17], trampolines use a dedicated syntax as opposed to giving programmers access to the full power of recursion.

4.3 Snapshot Initialisation of Trampolines

In Listing 1, the trampoline variable is initialised by a literal expression. Every deployment of `total` starts counting from 0. An alternative is to use **snapshot initialisation** where the "current value" of one of the input signals is used to initialise the trampoline variable at deployment-time. Listing 6 exemplifies this mechanism. In the definition of `min-max` two trampoline variables `i` and `a` are declared. Each variable remembers the smallest and largest value of the input signal `s` respectively. The point of the example is that `s` is used as the initialisation of trampoline variables `i` and `a`. Every deployment of the `min-max` reactor will take a "snapshot" of the signal `s`' current value in order to initialise its trampoline variables.

Snapshot initialisation can be considered as a non-essential feature. Indeed, reactors that require per-deployment initialisation can be easily transpiled into a more basic form. Haai features three-valued logic [8] which allows the programmer to use the primitive Boolean value `#u` (unknown) as a value to indicate that an actual value is missing. This value can be used as a literal initialisation value for trampoline variables. The reactor then needs to check using an `if`-test whether the value of the trampoline variable is equal to `#u` (using `unknown?`). For example: `(if (unknown? i) s i)`. Needless to say, the resulting reactor code would be be less elegant which explains why snapshot initialisation was conceived.

4.3.1 Example: `value-now`.

The ability to take a snapshot of a signal when deploying a trampoline reactor can be used to remember this value indefinitely. In many RP languages, there exists an operator which returns a signal's current value without introducing a dependency (e.g. the operator `value-now` in FrTime [14] and the method `now` in REScala [38]). Since Haai does not feature "primitive values" (everything is either a reactor or a signal), such an operation is meaningless. However, it *is* possible to create a new (constant) signal whose value is equal to the value of another signal's value at deployment-time. Listing 7 shows a trampoline reactor that creates such a signal. The signal `s` is only

```
(defr (value-now s | (init s))
  (out init | init))
```

Listing 7. Reactor producing the deployment-time value of a signal ad-infinitum.

```
(defr (monitor-turns sig control | (count 0))
  (def new-count (if control (+ count 1) 0))
  (out sig new-count | new-count))
```

Listing 8. A resettable turn counter for `sig`.

used to initialise the trampoline variable; it does not occur in the body of the reactor at all. As such all further values produced by `s` are ignored by the trampoline variable `init` which stays constant forever.

4.4 Turn Counting Semantics

Trampolines make it possible to refer to state from a previous turn. However, not every turn of a reactive program needs to make use of a trampoline variable's value. If none of the inputs of a deployment change, the trampoline variables contained therein do not need to be updated (i.e., a trampoline variable is only used by a deployment when a change occurs). We dub this the **turn counting semantics** of trampolines as only turns with changes to the inputs will use the trampoline variable, and consequently, update its value.

To illustrate, we implement a turn counter which tracks how often a given input signal `sig` changes. Listing 8 shows a reactor with two input variables (`sig` and `control`) and a trampoline variable `count` which is initialised to `0` for every deployment. It outputs a value based on `count` every time either `sig` or `control` changes. The `control` signal is used to instruct the deployment to either increment or reset `count` back to zero. As long as `control` stays `#t`, every change in `sig` will cause the reactor to increment and output `count`'s value. Hence the reactor is able to act as a per-deployment *turn counter*. As soon as `control` is set to `#f` the counter is reset back to `0`.

4.5 The Interactions of Trampoline Variables and Higher-Order Lexically Scoped Reactors

Recall from Section 3.3 that anonymous reactors (`rho`s) result in a signal that produces captures containing the `rho`'s DAG and lexically scoped signals. Haai's higher-order semantics will be used in Section 5.3 to create reactor implementations of `foldp` and `feedback` (from Table 1). To keep their explanations short, we explain the general semantics of trampoline variables and higher-order lexically scoped reactors here first.

4.5.1 `rho`s with Trampoline Variables. Anonymous reactors in Haai have access to signals from their lexical environment. When an anonymous reactor has a trampoline

```
(defr (example1 foo)
  (rho (bar | (acc 0))
    (def o (react-to foo bar acc))
    (out o | o)))
```

Listing 9. Anonymous reactor with a trampoline variable.

variable, its deployments can not only react when there's a change to any of the explicit sources of the anonymous reactor, but also if any of the captured signals from the lexical environment change their value. Listing 9 defines a reactor, called `example1`, which creates such an anonymous reactor. When `example1` is deployed (e.g, `(example1 sa)`), it creates a capture containing a reference to the deployment of `example1`. When that capture is then later deployed (e.g., `(c sb)`, assuming `c` is a signal that produced that capture), it is connected to the `sa` signal in its lexical environment and to the `sb` signal from its dynamic environment. Changes to either `sa` and `sb` cause the deployment of `c` to react. Thus, when `sa` produces a new value, the value of the trampoline variable must be propagated to `react-to` (whose implementation is, for the sake of the discussion, not important), even when `sb` did not produce a value in the same turn.

In essence, lexical signals can be considered as "implicit inputs" for capture deployments. Whenever they produce a value, the capture deployment must produce a new value and the trampoline variables of that capture deployment must be consequently used and updated as before. In other words, in the presence of higher-order lexically scoped reactors, our earlier definition of turn counting semantics (from Section 4.4) is not correct. It needs to be extended to also include the implicit inputs of any capture deployments.

4.5.2 Capturing a Trampoline Variable. Let us now study what happens when a `rho` refers to a trampoline variable located in its lexical scope instead of defining one in the anonymous reactor itself. Deployments of such an (anonymous) reactor must always use the latest value of the trampoline variable, even if there were no changes in the deployment where the trampoline variable was defined. Consider, for example, the implementation of `example2` in Listing 10. When `example2` is a deployed (e.g., `(example2 sa)`) it creates a capture (just as before). When that capture is later deployed (e.g., `(c sb)`, once again assuming `c` is a signal that produced that capture) it will depend not only on `sb`, but also on the trampoline variable `acc`. As `acc` is a trampoline variable, its value is only propagated through the deployment of the anonymous reactor if there is one input signal (including any captured, implicit input signal) changes. Thus, it is only when `bar` changes, that the value of the trampoline variable `acc` is propagated to the deployment of `react-to` (which for the sake of the discussion, has a different definition than the one in Section 4.5.1).

```
(defr (example2 foo | (acc 0))
  (def r (rho (bar) (react-to bar acc)))
  (def o (+ foo acc))
  (out r | o))
```

Listing 10. Anonymous reactor capturing a lexical trampoline variable.

```
(defr (example3 r | (acc 0))
  (def o (r acc))
  (out o | o))
```

Listing 11. A higher-order trampoline reactor that can, internally, deploy a capture which can be update independently from any of the sources of the trampoline reactor.

4.5.3 Deploying a Capture inside a Trampoline Reactor.

So far we have considered the two situations that may occur when an anonymous reactor uses a trampoline variable (either in its own environment, or from its lexical environment). The final interaction we have to consider is what happens when a trampoline reactor internally deploys a capture (with implicit inputs). The internal deployment of the capture may update, without there being any changes to the explicit (or implicit) inputs of the trampoline reactor itself and as such, the value of the trampoline variable must also be correctly propagated. Consider `example3` in Listing 11 which is a higher-order trampoline reactor. When `example3` is deployed using a capture (e.g., `(example3 (rho (in) (+ in sa)))`, where for the sake of the discussion, `sa` is defined elsewhere in the program and available in scope), the `example3` deployment internally deploys that capture. As `example3` passes the trampoline variable `acc` to the deployment of that capture, `acc`'s value must be propagated to this deployment even if no "inputs" of `example3` (to wit, `r`) produced a value.

4.5.4 Conclusion.

In short, a trampoline variable is not only used when one of the explicit inputs as present in the parameter list (Section 4.1) produces a value, but also if an implicit input produces a new value due to scope-driven reactivity (Section 4.5.1), which includes the implicit inputs (captured signals) of any capture deployments (Section 4.5.3). There is no distinction between trampoline variables defined in the local scope, and trampoline variables that exist in a lexical scope (Section 4.5.2).

5 Validation

In Section 2 we have argued that the list of stateful operators built into existing RP languages is ad-hoc and lacks a general underpinning. This section shows that trampoline variables can act as a general mechanism that allows us to express the inner workings of nearly every such operator. The structure of the section corresponds to the three categories of operators discerned in Table 1.

```
(defr (integral x | (prev-time time)
                    (acc 0))
  (def delta-t (- time prev-time))
  (def result (+ acc (* x delta-t)))
  (out result | time result))

(defr (derivative x | (prev-time time)
                      (prev-x x))
  (def delta-t (- time prev-time))
  (def delta-x (- x prev-x))
  (def result (if (> delta-t 0)
                  (/ delta-x delta-t)
                  0))
  (out result | time x))
```

Listing 12. Implementations of `integral` and `derivative`, based on trampoline variables.

5.1 Domain Specific Stateful Operators

We begin with the two mentioned domain specific operators from Table 1. Their implementation is shown in Listing 12. `integral` produces the Riemann integral of a signal with respect to `time`. `derivative` produces a numerical derivative.

`integral`. `integral` uses two trampoline variables. `acc` is initialised to `0` and is used to accumulate the value of the integral. `prev-time` is snapshot initialised to the current time. In each turn, the time elapsed since the previous turn is computed. This is subsequently used to compute the accumulated value which corresponds to `integral`'s output signal. This value and the current time are used to prepare the trampoline variables for the next turn.

`derivative`. `derivative` is similar from a Haai semantics point of view. Obviously, the mathematical formulas are different. If `delta-t` is equal to zero, no time has passed and as such no local derivative exists, in this situation the number 0 is produced on the output signal. This edge case is interesting: it occurs whenever the signal that corresponds to `x` produces values at a faster rate than `time`.

These two examples show that trampolines are a very elegant and simple technique to calculate some running quantity of a signal. Any quantity that combines the values sitting on a signal "one by one" can be easily programmed in the same fashion.

5.2 First-Order Stateful Operators

The implementations of the first-order stateful operators from Table 1 can be found in Listing 13. Some operators cannot be implemented in Haai. We have annotated them with a †.

```
(defr (pre s init | (acc init))
  (out acc | s))

(defr (latch s0 s1 | (s1-old s1)
                     (acc #u)
                     (old-out #u))
  (def tmp (if (eq? s1 s1-old) old-out acc))
  (def o (if (unknown? tmp) s1 tmp))
  (def new-acc (if (eq? s1 s1-old) acc s0))
  (out o | s1 new-acc tmp))

(defr (delay-by-turns s t init)
  (if (= t 0)
      s
      (pre (delay-by-turns s (- t 1) init)
           init)))
```

Listing 13. Implementation of `pre`, `latch` and `delay-by-steps`, based on trampoline variables.

`pre`. `pre` has one trampoline variable named `acc` that is snapshot initialised with `init`. The first value of `pre`'s output signal is equal to that trampoline variable. In every subsequent turn induced by a new value occurring on `s`, the trampoline variable is used as the reactor's output and the new value of `s` is stored in the trampoline variable for the next turn.

`last`†. The exact same semantics of Emfrp's [40] `@last` operator cannot be mimicked in Haai. In Emfrp, `@last` can only be used for signals whose definition is of the form `node init[`⟨value⟩`]` ⟨name⟩ `=` ⟨expression⟩ to provide an initial value in cases where it would otherwise be undefined. Notice however that goal of Emfrp's `last` is exactly the same as the aforementioned `pre` (which requires the user to specify an initial value in the deployment of `pre`). Hence, the inability to implement `last` is not an inherent weakness of trampolines.

`latch`. One difficulty with expressing `latch` is that Haai lacks the notion of "ticks" (cf. ActiveSheets [46]): a signal in Haai only "ticks" if its value changes between turns (unlike ActiveSheets where a cell, or formula, can also tick with its old value). To replicate the notion of a tick, our implementation of `latch` checks whether the current value of `s1` is equal to its old value (stored in the trampoline variable `s1-old`). If the value is different, then it uses the old value of `s0` as stored in the trampoline variable `acc` to determine its new output; otherwise, the output from a previous turn (as stored in the `old-out` trampoline variable) is produced as-is. However, if there was no prior output (e.g., during the deployment of `latch`) the trampoline variable `old-out` is equal to `#u` and in this case, the output signal of `latch` emits the current value of `s1`. The same test to check whether or not `s1` has changed is also used to update the `acc` trampoline variable such that the old value of `s0` is available the next time `s1` produces a value.

`hold`†. Haai lacks the distinction between continuous and discrete signals. Usually, `hold` is used to remember a discrete

```
(defr (foldp update init s | (acc init))
  (def o (update s acc))
  (out o | o))

(defr (feedback r init)
  (rho (a | (t init))
    (def (o1 o2) (r a t))
    (out o1 | o2)))
```

Listing 14. Implementations of `foldp` and `feedback`, based on trampoline variables.

signal's most recent value such that it can be used in later turns. This functionality already happens automatically for (Haai's) continuous signals. It is an open research question how trampoline variables interact with discrete signals.

`delay-by`†. Instead of implementing `delay-by` exactly as it is described in Table 1, we have implemented a variation called `delay-by-turns`. We first discuss it and then explain why it differs from `delay-by` in a rather fundamental way.

`delay-by-turns` delays a signal a number of turns. For example, the deployment expression `(delay-by-turns s 4 0)` corresponds to a signal which lags 4 successive turns behind of `s` (using `0` as the value produced by `delay-by-turns` until `s` has updated 4 times). In the implementation of `delay-by-turns` recursion is used to chain 4 different deployments of `pre`. Therefore, `delay-by-turns` itself does not use a trampoline variable but each inner deployment of `pre` (in each nested deployment of `delay-by-turns`) does.

This solution cannot be generalised to implement `delay-by` as `delay-by` relies on real-world time in a rather interesting way…Haai is very restrictive w.r.t. recursion: the implementation requires a constant number signal in any deployment expression of a recursive reactor which needs to decrease in value[4]. Any implementation of `delay-by` must therefore be able to dynamically allocate memory which is absent in Haai.

5.3 Higher-Order Stateful Operators

The implementations of the higher-order stateful operators can be found in Listing 14.

`foldp`. `foldp` has one accumulating trampoline variable `acc` which is snapshot initialised with the second input variable `init`. The trampoline variable is used in the internal deployment of the `update` reactor which reacts to the changes of `s` and `acc`. Each time when `s` changes its value, the new value is propagated through `update` together with the current value of `acc`. The result is produced on its sink signal and used as the updated accumulator.

`feedback`. `feedback` combines the expressive power of `loop` (e.g., from [31, 34]) and `pre`. The naive approach to first implement `loop`, and then combine it with our existing

[4]See [32] for more details on how to unwind recursion in RP programs.

implementation of `pre` does not work as `loop` requires cyclic reactors (which are not supported). Nonetheless, this did not prevent a working implementation of `feedback`. `feedback` has two parameters: `r` (which should be a signal producing reactor values) and `init` (which should be a signal producing the initial value). When `feedback` is deployed, an anonymous trampoline reactor is created. When this anonymous trampoline reactor is deployed (e.g., on a signal named `sa`), its trampoline variable `t` gets snapshot initialised to `init` (from its lexical scope) and the reactors produced by `r` (also from its lexical scope) are deployed on `sa` and `t`. The first sink signals of `(r a t)` is used as the anonymous reactor's own sink signal and the second sink signal is used to update the trampoline variable.

`feedback` is an extremely powerful construct as it allows one to express a stateful computation without using trampolines, and "tighten the knot" in another part of a program. In other words, `feedback` gives programmers more control over in which deployment a trampoline variable is stored.

5.4 Conclusion

This section has shown that trampoline variables are sufficiently general to serve as an underpinning for nearly all the ad-hoc operators discussed in Table 1. Apart from `last` (which is intricately intertwined with Emfrp's technicalities) and `hold` (which only make sense for discrete signals), `delay-by` is the only stateful operator whose semantics cannot be expressed using trampolines. This is because the amount of memory occupied by trampolines is constant whereas the amount of memory required by `delay-by` depends on the specified (real) time as well as on the pace of the signal that is being delayed for that time.

6 Hot-Swapping with Trampolines

Haai as a language is part of a larger project. The goal is to understand reactive systems in terms of reactions "all the way down". In the long run, the goal is to conceive the IDE as a reactive system as well, similarly to how SmallTalk [21] and Self [44] are objects "all the way down". This means that it will be necessary at some point to replace a part of a running system, a feature known as *hot-swapping*. Reactors form the obvious level of granularity for this. Briefly explained, a reactor can be hot-swapped by replacing all current deployments with a deployment of their new behaviour. For stateful reactors this raises the question whether or not the state accumulated by the deployment of the old behaviour can be handed over to the deployment of the new behaviour. We show that the disciplined state management syntax of trampoline reactors allows a programmer to gain more control over this.

The main insight that shows the use of trampoline variables when hot-swapping is that every stateful computation is required to store any stateful data in a trampoline variable: no state is hidden by a built-in operator. We defined three hot-swapping mechanisms (of which two are able to retain state using trampoline variables):

1. **Without State Retention** When a reactor is hot-swapped with a new behaviour, all trampoline variables are initialised as if the deployment had just been created. Hot-swapping without state retention is useful if the old accumulated state is incompatible between two code versions (e.g., due to drastic changes to the internal data representation that cannot be reconciled, or because the trampoline variables contains incorrect values which cannot be re-used in the new code version).
2. **With By-Name State Retention** The simplest mechanism for state retention of trampoline reactor deployments is to use the names of the trampoline variables to determine whether or not any state can be retained. Briefly explained: if the new version of a reactor has a trampoline variable that has the same name as a trampoline variable in the old version of the code, the value held by the old trampoline variable can be used as the initial value of the new trampoline variable (the initial value written in the trampoline declaration is thus ignored for all existing deployments of the hot-swapped trampoline reactor). Only trampoline variables which do not find a corresponding trampoline variable in the previous deployment are initialised using the ordinary trampoline initialisation semantics. This state retention mechanism is, in general, only usable if the values of the old trampoline variables contain the same kind of information (i.e., the same type and using the same measurement unit).
3. **With an Explicit State Transformer** In more complex situations, the by-name state retention mechanism is not powerful enough. If the new implementation of a reactor uses a different data representation or measurement unit (e.g., using milliseconds instead of seconds) for the trampoline variables, or has renamed them, the new deployment may not be correctly initialised. By supplying a state transformer, a programmer has precise control on how the new trampoline variables are initialised.

6.1 Hot-Swapping in Practice

An interactive session of Haai's command-line interpreter presenting the two state retention mechanisms is shown in Listing 15. Black lines indicate printed output, green lines indicate program definitions, and purple lines indicate the usage of interpreter commands (which are explained below).

Lines 1–6 show a first implementation of `average` which should compute a running average of a signal (`x`) over time (`t`). Each time `x` changes, it is expected that `t` contains a timestamp (e.g., expressed in seconds, such as `time`) that denotes when the signal's value has changed. It is expected that between two temperature readings, `x`'s value remains constant.

```
 1 (defr (average x t | (acc 0) (old-x x) (old-t t) (total-t 0))
 2   (def delta-t (- t old-t))
 3   (def new-acc (+ acc (* old-x delta-t)))
 4   (def new-total-t (+ total-t delta-t))
 5   (def avg (if (= total-t 0) 0 (/ acc total-t)))
 6   (out avg | new-acc x t new-total-t))
 7 :new-signal temperature 20
 8 :new-signal seconds 0
 9 (def avg-temperature (average temperature seconds))
10 :monitor avg-temperature
11 Value of ‘avg-temperature‘ = 0
12 :set temperature 22 seconds 1
13 :set temperature 24 seconds 2
14 Value of ‘avg-temperature‘ = 20.0
15 (defr (average x t | (acc 0) (old-x x) (old-t t) (total-t 0))
16   (def delta-t (- t old-t))
17   (def new-acc (+ acc (* old-x delta-t)))
18   (def new-total-t (+ total-t delta-t))
19   (def avg (if (= new-total-t 0) 0 (/ new-acc new-total-t)))
20   (out avg | new-acc x t new-total-t))
21 :set temperature 28 seconds 6
22 Value of ‘avg-temperature‘ = 23.0
23 :set-transformer average (rho (a b c d) (out (/ a d) b c))
24 (defr (average x t | (acc 0) (old-x x) (old-t t))
25   (def avg
26     (if (= t 0)
27         0
28         (/ (+ (* acc old-t) (* old-x (- t old-t))) t)))
29   (out avg | avg x t))
30 :set temperature 26 seconds 10
31 Value of ‘avg-temperature‘ = 25.0
```

Listing 15. Changing the behaviour of a reactor at-runtime in Haai's console-based interpreter.

To aid in our demonstration of Haai's ability to hot-swap reactors, we use the interpreter's ability to create so-called *mock signals*. Mock signals can be updated manually by the programmer and allows one to interactively experiment with a reactor's behaviour. On Lines 7–8 we use the interpreter command `:new-signal` twice to create two mock signals named `temperature` and `seconds` which are initialised to 20 (℃) and 0 (seconds) respectively.

The `average` reactor is deployed on `temperature` and `seconds` on Line 9, creating a signal named `avg-temperature`. The interpreter command `:monitor` is used (Line 10) to monitor which values it produces. It immediately prints `avg-temperature`'s latest value and will print its value every time when another value is produced during the interactive session (instead of having to `:print` each time we are interested in a signal's last value). `avg-temperature`'s initial value is 0 as so far no time has passed according to the `average` deployment (Line 11).

The current implementation of `average` contains a small bug in which the output is not correctly computed. For example, if one second after `average` has been deployed the temperature changes to 22, we would assume that `avg-temperature` would produce a 20 (as in the first second, `temperature` remained a constant 20℃). However, if we change `temperature`'s and `seconds`'s values to 22 and 1 respectively[5] (Line 12), nothing is printed: `avg-temperature`'s value thus did not change. However, if we change `temperature` and `seconds` again, e.g., to respectively 24 and 2 (Line 13), an updated value is printed (Line 14) which is the value we expected previously. It seems that `avg-temperature` lags behind one turn with respect to the actual changes that should be processed.

[5]The `:set` command can change multiple mock signals at once by alternating the names of the mock signals with their corresponding new values.

6.1.1 Fixing `average`.

This bug is caused by a mistake on Line 7. Instead of using the "new" values of the trampoline variables `total-t` and `acc`, the old values are used to calculate the new average. As it is not possible to use the new values of a trampoline variable (as trampoline variables are always updated at the end of a turn), the program must compute the `avg` in terms of `new-total-t` and `new-acc`.

We apply this bug fix by changing the implementation of `average` on Lines 15–20. By default, Haai will use the by-name state retention mechanism to hot-swap a reactor in a running program. The next time when a new temperature reading is processed by the system, which we emulate by changing the mock signals `temperature` and `seconds` to 28 and 6 respectively (Line 21), the new implementation of `average` is used which now produces the correct output (Line 22), to wit $^{20\cdot1+22\cdot1+24\cdot4}/_{6} = 23$ (it was 20℃ for 1 second, 22℃ for 1 second, and 24℃ for 4 seconds, thus the average temperature is 23℃).

6.1.2 Refactoring `average`.

To showcase how an explicit state transformer can be used, we decided to make further changes to `average`. Currently, the trampoline variable `total-t` is always equal to `t` and as such there is no need for `total-t`. And, instead of storing a "weighted sum" in `acc`, the average can be stored in `acc` instead. To compute the next average, the old "weighted sum" can be computed by multiplying `acc` (the old average) and `old-t` (the old time duration). The modified implementation of `average` is shown on Lines 24–29. Since the meaning of the values stored in `acc` has changed, the default by-name state retention mechanism is not sufficient to correctly retain state.

An explicit state transformer can be passed using the `:set-transformer` interpreter command[6] (Line 23), which needs to be set before the new reactor definition is supplied to the interpreter. Each state transformer is only used once (i.e. the next hot-swap of `average` would use the by-name state retention mechanism again). On Line 30 we change the values of the `temperature` and `seconds` for the last time, and the `avg-temperature` signal consequently changes to $^{20\cdot1+22\cdot1+24\cdot4+28\cdot4}/_{10} = 25$ (Line 31), showing that the values of the old trampoline variables were correctly used to initialise the new deployment of `average`.

[6]As Haai lacks functions, a reactor (e.g., one implemented with `rho`) is used for the state transformer. This meta reactor is only used once, for every existing deployment, and thus behaves as if it would be a function.

7 Discussion and Related Work

7.1 Generalising State Retention

Trampoline variables alone are not yet powerful enough to generalise state retention. Consider, for example, the following definition of `offset` which calculates the difference of a numerical signal with respect to its initial value:

```
(defr (offset x)
  (- x (value-now x)))
```

While this reactor does not define any trampoline variables, its deployments are stateful due to the inner deployment of `value-now`. When `offset` is modified (e.g., by refactoring it, or by adding new functionality) all existing deployments of `offset` will lose their state. When the new behaviour is being deployed, the new deployments of `value-now` have no way to restore their previous state, as in general, the new implementation of `offset` may have relocated their position in the DAG. While one could assume that, for certain code changes, that the new `value-now` deployment is equivalent with respect to the old deployment, it is hard to assume this in general. This could be solved by making more advanced state transformers that can "traverse" the deployment hierarchy, but we regard this approach as being too counter-intuitive (especially considering Haai's dynamic deployment expressions which make it less obvious which deployment expressions can contain state).

Other RP languages with support for hot-code reloading also have similar issues related to state retention:

- In Midair [30], an RP language with support for hot-code reloading built on top of Haskell, `SFlows` (i.e. signal functions from [22]) can be replaced at-runtime by the programmer. To hot-swap a reactive program in Midair, all existing instances of a named `SFlow` are replaced with a new `SFlow` object that is returned by a hot-swap function. Instead of giving this function access to any of the stateful variables contained in an existing instance of an `SFlow`, the function only gets access to the current source and sink signal values (by supplying both as arguments to the hot-swap function). The hot-swap function is applied to each instantiated `SFlow` separately (like our state transformers are for each existing deployment of the modified reactor), and can return a different `SFlow` object each time (unlike our state transformers which can only override the initial values of the trampoline variables). This mechanism makes it possible to initialise the inner state, as long as it can be derived from the current input and output values and is therefore not as powerful as trampolines.
- Instead of replacing only named components of a dependency graph (like in Haai or Midair) a whole RP program can be replaced at once. For example in essence-of-live-coding [6], a live coding framework for Haskell on which an RP library has been built, where the stateful information of a (running) RP program (as a whole) closely maps onto the structure of its dependency graph. When the program changes, this state is automatically migrated (by taking into account the type information of the old and the new program, by using generics from [27]) such that the information held in the old state, is migrated for the new program (assuming it is compatible). While easy to understand for basic code changes that respect the structure of the dependency graph, more complex code changes are outright impossible as there is no way to override the automatic state migration. Trampoline variables, on the other hand, give direct access to the stateful values held by the stateful reactor that is being hot-swapped.

We therefore conclude that state retention for RP languages does not have a commonly-accepted solution yet and consider this as our future work.

7.2 Alternatives to Trampoline Variables

Besides lifting impure functions and built-in stateful operators (as explained in Section 2) there have been other approaches proposed for managing state in RP languages.

- In E-FRP [49], signals are updated by an event handler which uses the signal's current value (together with the values of other signals) to determine the new value of the signal. In this approach, every signal defined using event handlers is essentially stateful. Unlike Haai, E-FRP has no support for recursion nor for switching (e.g., dynamic deployments or an explicit `switch` operator) and is therefore much more restricted.
- Instead of storing stateful information in an operator, a signal itself could automatically store all its past values, This approach is used in [25], where the history of a *persistent* signals are stored in a time-series database. There is no need for operators like `s.avg()` (which computes the average of a persistent signal `s`) to remember any history (or accumulation variable) themselves if it can be derived from the persistent signal's stored history.

8 Conclusion

This paper presented *trampoline variables*, a generalised approach for stateful variables which are integrated directly in the RP language itself. We have described their semantics in a higher-order RP language called Haai and used them to implement all kinds of well-known stateful operators that can be found in other RP languages. In addition, they force RP programmers to explicitly manage state, which makes it possible to seamlessly handle state migrations between two different code versions when hot-swapping.

Acknowledgments

Bjarno Oeyen and Sam Van den Vonder are funded by the Research Foundation - Flanders (FWO) under grant number 1S93820N1 and 1S95318N respectively.

References

[1] [n.d.]. Manual - REScala. https://web.archive.org/web/20210803092716/http://www.rescala-lang.com/manual. Accessed: 2021-08-03.

[2] [n.d.]. RxJS - API List. https://web.archive.org/web/20210804174704/https://rxjs.dev/api/. Accessed: 2021-08-04.

[3] [n.d.]. statefulMapConcat • Akka Documentation. https://web.archive.org/web/20210804174155/https://doc.akka.io/docs/akka/current/stream/operators/Source-or-Flow/statefulMapConcat.html. Accessed: 2021-08-04.

[4] N. I. Adams, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, G. J. Sussman, M. Wand, and H. Abelson. 1998. Revised^5 Report on the Algorithmic Language Scheme. *SIGPLAN Not.* 33, 9 (Sept. 1998), 26–76. https://doi.org/10.1145/290229.290234

[5] Joe Armstrong. 2010. Erlang. *Commun. ACM* 53, 9 (Sept. 2010), 68–75. https://doi.org/10.1145/1810891.1810910

[6] Manuel Bärenz. 2020. The essence of live coding: change the program, keep the state!. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, REBLS@SPLASH 2020* (Virtual, USA) *(REBLS@SPLASH 2020)*. ACM, New York, NY, USA, 2–14. https://doi.org/10.1145/3427763.3428312

[7] Manuel Bärenz and Ivan Perez. 2018. Rhine: FRP with type-level clocks. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018*, Nicolas Wu (Ed.). ACM, 145–157. https://doi.org/10.1145/3242744.3242757

[8] Merrie Bergmann. 2008. *An introduction to many-valued and fuzzy logic : semantics, algebras, and derivation systems.* Cambridge University Press, Cambridge New York.

[9] Stephen Blackheath and Anthony Jones. 2016. *Functional reactive programming.* Manning Publications Co.

[10] Guerric Chupin and Henrik Nilsson. 2019. Functional Reactive Programming, restated. In *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, Ekaterina Komendantskaya (Ed.). ACM, 7:1–7:14. https://doi.org/10.1145/3354166.3354172

[11] Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. 2006. Mixing signals and modes in synchronous data-flow systems. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software, EMSOFT 2006, October 22-25, 2006, Seoul, Korea*, Sang Lyul Min and Wang Yi (Eds.). ACM, 73–82. https://doi.org/10.1145/1176887.1176899

[12] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. 2017. SCADE 6: A formal language for embedded critical software development (invited paper). In *11th International Symposium on Theoretical Aspects of Software Engineering, TASE 2017, Sophia Antipolis, France, September 13-15, 2017*, Frédéric Mallet, Min Zhang, and Eric Madelaine (Eds.). IEEE Computer Society, 1–11. https://doi.org/10.1109/TASE.2017.8285623

[13] Gregory Harold Cooper. 2008. *Integrating dataflow evaluation into a practical higher-order call-by-value language.* Ph.D. Dissertation. Brown University.

[14] Gregory H. Cooper and Shriram Krishnamurthi. 2006. Embedding Dynamic Dataflow in a Call-by-Value Language. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3924)*, Peter Sestoft (Ed.). Springer, 294–308. https://doi.org/10.1007/11693024_20

[15] Antony Courtney. 2001. Frappé: Functional Reactive Programming in Java. In *Practical Aspects of Declarative Languages, Third International Symposium, PADL 2001, Las Vegas, Nevada, USA, March 11-12, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 1990)*, I. V. Ramakrishnan (Ed.). Springer, 29–44. https://doi.org/10.1007/3-540-45241-9_3

[16] Evan Czaplicki and Stephen Chong. 2013. Asynchronous functional reactive programming for GUIs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 411–422. https://doi.org/10.1145/2491956.2462161

[17] Sam Van den Vonder, Thierry Renaux, Bjarno Oeyen, Joeri De Koster, and Wolfgang De Meuter. 2020. Tackling the Awkward Squad for Reactive Programming: The Actor-Reactor Model. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 19:1–19:29. https://doi.org/10.4230/LIPIcs.ECOOP.2020.19

[18] Conal Elliott. 1998. Functional implementations of continuous modeled animation. In *Principles of Declarative Programming*. Springer, 284–299.

[19] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997*, Simon L. Peyton Jones, Mads Tofte, and A. Michael Berman (Eds.). ACM, 263–273. https://doi.org/10.1145/258948.258973

[20] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. Frenetic: a network programming language. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). ACM, 279–291. https://doi.org/10.1145/2034773.2034812

[21] Adele Goldberg. 1983. *Smalltalk-80 : the language and its implementation.* Addison-Wesley, Reading, Mass.

[22] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. 2002. Arrows, Robots, and Functional Reactive Programming. In *Advanced Functional Programming, 4th International School, AFP 2002, Oxford, UK, August 19-24, 2002, Revised Lectures (Lecture Notes in Computer Science, Vol. 2638)*, Johan Jeuring and Simon L. Peyton Jones (Eds.). Springer, 159–187. https://doi.org/10.1007/978-3-540-44833-4_6

[23] Graham Hutton. 1999. A Tutorial on the Universality and Expressiveness of Fold. *J. Funct. Program.* 9, 4 (1999), 355–372.

[24] Daniel Ignatoff, Gregory H. Cooper, and Shriram Krishnamurthi. 2006. Crossing State Lines: Adapting Object-Oriented Frameworks to Functional Reactive Languages. In *Functional and Logic Programming, 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3945)*, Masami Hagiya and Philip Wadler (Eds.). Springer, 259–276. https://doi.org/10.1007/11737414_18

[25] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. 2021. Signal Classes: A Mechanism for Building Synchronous and Persistent Signal Networks. In *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference) (LIPIcs, Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 17:1–17:30. https://doi.org/10.4230/LIPIcs.ECOOP.2021.17

[26] Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 2016. 43 years of actors: a taxonomy of actor models and their key properties. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE 2016, Amsterdam, The Netherlands, October 30, 2016*, Sylvan Clebsch, Travis Desell, Philipp Haller, and Alessandro Ricci (Eds.). ACM, 31–40. https://doi.org/10.1145/3001886.3001890

[27] Ralf Lämmel and Simon L. Peyton Jones. 2003. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of TLDI'03: 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, New Orleans, Louisiana, USA, January 18, 2003*, Zhong Shao and Peter Lee (Eds.). ACM, 26–37. https://doi.org/10.1145/604174.604179

[28] Geoffrey Mainland, Greg Morrisett, and Matt Welsh. 2008. Flask: staged functional programming for sensor networks. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, James Hook and Peter Thiemann (Eds.). ACM, 335–346. https://doi.org/10.1145/1411204.1411251

[29] Leo A. Meyerovich, Arjun Guha, Jacob P. Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: a programming language for Ajax applications. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, Shail Arora and Gary T. Leavens (Eds.). ACM, 1–20. https://doi.org/10.1145/1640089.1640091

[30] Tom E. Murphy. 2016. A livecoding semantics for functional reactive programming. In *Proceedings of the 4th International Workshop on Functional Art, Music, Modelling, and Design, FARM@ICFP 2016, Nara, Japan, September 24, 2016*, David Janin and Michael Sperber (Eds.). ACM, 48–53. https://doi.org/10.1145/2975980.2975986

[31] Henrik Nilsson, Antony Courtney, and John Peterson. 2002. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. 51–64.

[32] Bjarno Oeyen, Sam Van den Vonder, and Wolfgang De Meuter. 2020. Reactive Sorting Networks. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, REBLS@SPLASH 2020* (Virtual, USA) *(REBLS@SPLASH 2020)*. ACM, New York, NY, USA, 38–50. https://doi.org/10.1145/3427763.3428316

[33] Bjarno Oeyen, Humberto Rodríguez-Avila, Sam Van den Vonder, and Wolfgang De Meuter. 2018. Composable higher-order reactors as the basis for a live reactive programming environment. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, REBLS@SPLASH 2018, Boston, MA, USA, November 4, 2018*, Guido Salvaneschi, Wolfgang De Meuter, Patrick Eugster, Lukasz Ziarek, and Francisco Sant'Anna (Eds.). ACM, 51–60. https://doi.org/10.1145/3281278.3281284

[34] Ross Paterson. 2001. A New Notation for Arrows. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001*, Benjamin C. Pierce (Ed.). ACM, 229–240. https://doi.org/10.1145/507635.507664

[35] Ivan Perez, Manuel Bärenz, and Henrik Nilsson. 2016. Functional reactive programming, refactored. In *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*, Geoffrey Mainland (Ed.). ACM, 33–44. https://doi.org/10.1145/2976002.2976010

[36] Bob Reynders and Dominique Devriese. 2017. Efficient Functional Reactive Programming Through Incremental Behaviors. In *Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10695)*, Bor-Yuh Evan Chang (Ed.). Springer, 321–338. https://doi.org/10.1007/978-3-319-71237-6_16

[37] Bob Reynders, Frank Piessens, and Dominique Devriese. 2020. Gavial: Programming the web with multi-tier FRP. *Art Sci. Eng. Program.* 4, 3 (2020), 6. https://doi.org/10.22152/programming-journal.org/2020/4/6

[38] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014. REScala: bridging between object-oriented and functional style in reactive applications. In *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014*, Walter Binder, Erik Ernst, Achille Peternier, and Robert Hirschfeld (Eds.). ACM, 25–36. https://doi.org/10.1145/2577080.2577083

[39] Abhiroop Sarkar and Mary Sheeran. 2020. Hailstorm: A Statically-Typed, Purely Functional Language for IoT Applications. In *PPDP '20: 22nd International Symposium on Principles and Practice of Declarative Programming, Bologna, Italy, 9-10 September, 2020*. ACM, 12:1–12:16. https://doi.org/10.1145/3414080.3414092

[40] Kensuke Sawada and Takuo Watanabe. 2016. Emfrp: a functional reactive programming language for small-scale embedded systems. In *Companion Proceedings of the 15th International Conference on Modularity, Málaga, Spain, March 14 - 18, 2016*, Lidia Fuentes, Don S. Batory, and Krzysztof Czarnecki (Eds.). ACM, 36–44. https://doi.org/10.1145/2892664.2892670

[41] Kazuhiro Shibanai and Takuo Watanabe. 2018. Distributed functional reactive programming on actor-based runtime. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE!@SPLASH 2018, Boston, MA, USA, November 5, 2018*, Joeri De Koster, Federico Bergenti, and Juliana Franco (Eds.). ACM, 13–22. https://doi.org/10.1145/3281366.3281370

[42] Artur Sterz, Matthias Eichholz, Ragnar Mogk, Lars Baumgärtner, Pablo Graubner, Matthias Hollick, Mira Mezini, and Bernd Freisleben. 2021. ReactiFi: Reactive Programming of Wi-Fi Firmware on Mobile Devices. *Art Sci. Eng. Program.* 5, 2 (2021), 4. https://doi.org/10.22152/programming-journal.org/2021/5/4

[43] Kohei Suzuki, Kanato Nagayama, Kensuke Sawada, and Takuo Watanabe. 2018. CFRP: A functional reactive programming language for small-scale embedded systems. In *Theory and Practice of Computation: Proceedings of Workshop on Computation: Theory and Practice WCTP2016*. World Scientific, 1–13.

[44] David M. Ungar and Randall B. Smith. 2007. Self. In *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007*, Barbara G. Ryder and Brent Hailpern (Eds.). ACM, 1–50. https://doi.org/10.1145/1238844.1238853

[45] Atze van der Ploeg and Koen Claessen. 2015. Practical principled FRP: forget the past, change the future, FRPNow!. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, Kathleen Fisher and John H. Reppy (Eds.). ACM, 302–314. https://doi.org/10.1145/2784731.2784752

[46] Mandana Vaziri, Olivier Tardieu, Rodric Rabbah, Philippe Suter, and Martin Hirzel. 2014. Stream Processing with a Spreadsheet. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8586)*, Richard E. Jones (Ed.). Springer, 360–384. https://doi.org/10.1007/978-3-662-44202-9_15

[47] Andreas Voellmy and Paul Hudak. 2011. Nettle: Taking the Sting Out of Programming Network Routers. In *Practical Aspects of Declarative Languages - 13th International Symposium, PADL 2011, Austin, TX, USA, January 24-25, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6539)*, Ricardo Rocha and John Launchbury (Eds.). Springer, 235–249. https://doi.org/10.1007/978-3-642-18378-2_19

[48] Zhanyong Wan, Walid Taha, and Paul Hudak. 2001. Real-Time FRP. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001*, Benjamin C. Pierce (Ed.). ACM, 146–156. https://doi.org/10.1145/507635.507654

[49] Zhanyong Wan, Walid Taha, and Paul Hudak. 2002. Event-Driven FRP. In *Practical Aspects of Declarative Languages, 4th International Symposium, PADL 2002, Portland, OR, USA, January 19-20, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2257)*, Shriram Krishnamurthi and C. R. Ramakrishnan (Eds.). Springer, 155–172. https://doi.org/10.1007/3-540-45587-6_11

[50] Sheng Wang and Takuo Watanabe. 2019. Functional reactive EDSL with asynchronous execution for resource-constrained embedded systems. In *International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*. Springer, 171–190.