Composable Higher-Order Reactors as the Basis for a Live Reactive Programming Environment

Bjarno Oeyen
Vrije Universiteit Brussel
Brussels, Belgium
boeyen@vub.be

Humberto Rodriguez Avila
Vrije Universiteit Brussel
Brussels, Belgium
rhumbert@vub.be

Sam Van den Vonder
Vrije Universiteit Brussel
Brussels, Belgium
svdvonde@vub.be

Wolfgang De Meuter
Vrije Universiteit Brussel
Brussels, Belgium
wdmeuter@vub.be

Abstract

A live programming environment allows programmers to edit programs while they are running. This means that successive "edit steps" must not allow a programmer to bring the program in a form that does not make any sense to the underlying language processor (i.e., parser, compiler,...). Many live programming environments therefore rely on disciplined edit steps that are based on language elements such as objects, classes, and methods. Textual modifications to these elements are not seen as edit steps until some "accept" button is hit. Unfortunately, no such elements exist in current reactive languages. We present a new reactive language, called Haai, that is based on first-class higher-order *reactors*. Linguistically, Haai programs correspond to reactors or compositions of reactors. At run-time, reactors produce an infinite stream of values just like signals and behaviours in existing languages. Haai's live programming environment relies on textual modifications of entire reactors as its basic edit steps. Changing a reactor automatically updates all occurrences of that reactor in the reactive program, while it is running.

CCS Concepts • **Software and its engineering → Data flow languages**; **Integrated and visual development environments**;

Keywords Reactive Programming, High-Order Programming, Live Programming

ACM Reference Format:
Bjarno Oeyen, Humberto Rodriguez 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 '18), November 4, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3281278.3281284

REBLS '18, November 4, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6070-8/18/11…$15.00
https://doi.org/10.1145/3281278.3281284

1 Introduction

Live Programming has received a lot of attention recently. The main idea is to have a development environment that allows a programmer to edit bits and pieces of a program while it is running. Every live programming environment is built atop some programming paradigm: one can build live programming environments for object-oriented programming (e.g., Self [10]), imperative programming (e.g., LightTable [1]),.... In this paper, we contribute to the realm of *Live Reactive Programming*. We aim to build a programming environment that allows one to modify reactive programs while they are reacting.

Any live programming environment has to define those program elements whose modifications will be considered as valid "edit steps". For example, in Self or Smalltalk, the elements modified by an edit step correspond to individual objects and classes. In other existing live programming environments, edit steps correspond to individual keystrokes – i.e., every time the live programmer hits a key, the program is updated [12, 13]. Optimally, edit steps should exhibit transactional behaviour: before and after each edit step, the program should be in a valid form that allows the runtime to hot-swap the original version of the program with the modified version. This means that character-level edit steps are undesirable because adding or removing individual characters typically brings the program in a form that cannot be processed by the language processor (i.e., parser, compiler, ...). A programming language that allows such fine-grained changes results in programs behaving unpredictably once there are syntax errors in the program text. Languages like Self avoid this problem by defining "edit steps" based on successive clicks on the "doIt" or "acceptIt" button. In this way, the programmer has more control over when the live environment changes the running program. Invalid code, caused by intermediate keystrokes is never considered as an edit step.

Current reactive programming languages lack the kind of linguistic components that play the *code-organising role* that is played by objects in Self and classes in Smalltalk.

They either consider every single keystroke as an "edit step" (possibly leaving the program in an invalid format) or they require to hot-swap the entire program based on a global "compile" button (as in Elm [6]). One might argue that the signals that constitute a reactive program could play this role as they can be implemented as first-class values. However, in current languages signals only exist in the language runtime. They have no linguistic counterpart in the programming environment apart from the strings of characters that make up the identifiers that are used to describe them. For example, programmers cannot "grab" or "point to" signals as manipulable entities.

In this paper, we present a new reactive programming language (called Haai[1]) and a supporting live programming environment (called SharkTank). The central linguistic concept of Haai is the **reactor**, which is used in the design of the language as a way of composing reactors out of other reactors and is used for changes in the live programming environment. SharkTank will be able to change the behaviour of reactors while they are being reacting, such that the modified behaviour is used in all locations where the reactor is being used. Textual changes made to reactors (delimited by successive clicks on the "Commit" button) form the edit steps of SharkTank, just as in Self or Smalltalk. After successfully checking the syntax of the modified reactor (and compiling it), the modified version is hot-swapped in the Haai runtime. From that moment on, all other reactors that depend on the modified reactor automatically use the modified version. The runtime environment for Haai can be seen as a reactive environment that reacts to both base events (those created by the program itself), and code change events (those created by the programming environment).

Haai has the following properties:

1. First-class **reactors** are the basis of reactivity. Reactors roughly correspond to the behaviours that exist in other reactive languages. Reactors can be programmed textually and given a name at the level of the programming environment.
2. Reactors can be composed resulting in more reactors. Haai features both pointwise as well as point-free composition. The latter corresponds to applying a set of (built-in) **composition operators** which "glue together" existing reactors. The former corresponds to programming reactors whose code "calls" **parametrised reactors** with streams as "arguments".
3. Apart from taking reactors as parameters, reactors can also emit reactors as values. This gives rise to **higher-order reactors**. Haai's built-in point-free composition operators are nothing but higher-order reactors that glue together their argument reactors. The reactors that are emitted as values by a reactor are textually denoted by means of **anonymous reactors** in the same way that higher-order functions can return functions by returning a so-called `lambda` expression. In Haai, `rho` expressions generate anonymous reactors.
4. Many reactive programming languages rely on a host language (e.g., Scala or Haskell) or contain a non-reactive functional sublanguage (e.g., Elm). Haai, on the other hand, is "reactors all the way down". For example, reactors never "call" auxiliary functions but rather "push" values to auxiliary reactors. Even primitive operators such as `+` are reactors.

The rest of the paper is structured as follows. Section 2 introduces the concept of reactors for programming reactive programs. Section 3 expands upon these concepts and shows how reactors can be used for higher-order reactive programming. Section 4 shows how reactors can be used for live programming, and in Section 5 we discuss related work.

2 Reactors as Reactive Programs

Traditional reactive programming languages consist of a reactive layer which allows the programmer to think in terms of time-varying values (typically called signals or behaviours) and a functional layer which requires a programmer to think in terms of functions. "Lifting" (either explicitly or implicitly) is the concept that connects between both layers: a lifted function that is "applied" to a behaviour creates a new behaviour which then internally applies the function (with traditional call semantics) to all values that are emitted by the argument behaviour.

Haai breaks with this tradition. Haai only has a reactive layer. It does so by replacing functions by so-called **reactors** which can be first-class values. They are used to structure a reactive program, but they can also serve as the values that are emitted by these reactive programs. Each reactor can be thought of as a fully-fledged reactive program. Another way to think of a reactor is to think of them as an abstraction over a dependency graph.

2.1 Reactors by Example

Haai uses a Scheme-like syntax[2]. Syntactically reactors are very similar to function definitions in Scheme. A reactor is defined by `define-reactor` and has a name, formal parameters (the "input" of the reactor), and a body that defines its functionality. Listing 1 defines a reactor `rectangle` that calculates the area and the perimeter of a rectangle given its width and height. Informally, each time a value arrives on one of the inputs `w` or `h`, the reactor will calculate the `area` and the `perimeter` and then emit both values to whomever is listening to the reactor. Every reactor can be thought of as a stand-alone reactive program. Its corresponding dependency graph, a directed acyclic graph (DAG), is shown in Figure 1. The sources of the graph correspond to the formal parameters of the reactor, the sinks of the graph correspond

[1] Haai means Shark in Dutch and is pronounced as "high".

[2] A prototypical Haai interpreter was implemented in Racket.

```
1 (define-reactor (rectangle w h)
2   (define area (* w h))
3   (define perimeter (+ (* w 2)
4                        (* h 2)))
5   (export area perimeter))
```

Listing 1. Reactor that calculates the area and perimeter of a rectangle.

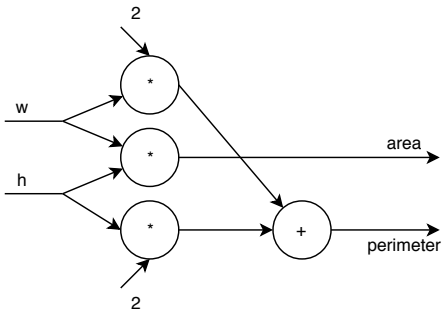


Figure 1. The directed acyclic graph showing how data flows from input to output of the reactor in Listing 1.

to the exported values, and the internal nodes correspond to reactive expressions that make up the body of the reactor.

2.2 Deploying Reactors

Reactors correspond to *inactive* reactive programs because reactors do not do anything as long as their formal parameters have not been "connected" to some active source of values. That is the role played by **streams** in Haai. Streams are "living" in the interpreter and perpetually emit values. Streams are either built-in (e.g., `time`) or are the result of evaluating the expressions exported by a reactor. Central to this is the notion of deploying reactors.

We say that a reactor is **deployed** if its input parameters are "connected" to already existing streams. The result of deploying the reactor is one or more new streams that correspond to the reactor's `export` expression. For instance, if we consider the built-in streams `mouse-x` and `mouse-y`, then the expression

```
(rectangle mouse-x mouse-y)
```

is a deployment of reactor `rectangle` that will create two new streams that will produce the areas and the perimeters of the rectangles defined by the position of the mouse w.r.t. the origin. Notice that deploying reactors follows the same syntax as calling procedures in Scheme[3].

Conceptually, every deployment of a reactor to a number of existing streams creates an instance of the reactor (i.e., a copy of the reactor's DAG). The streams created in that copy

[3] Some Haai expressions of the form (F a1 ... an) use a preserved keyword for `F` (such as `if`, `rho`, `past`, `define` and `define-reactor`). These are handled by the compiler in a special way and we call them special forms as in Scheme.

```
1 (define-reactor (rectangle-scale w h factor)
2   (define w-s (scale w factor))
3   (define h-s (scale h factor))
4   (define (a p) (rectangle w-s h-s))
5   (export a p))
```

Listing 2. Reactor definition that scales a rectangle before calculating the area and perimeter.

(some of which are exported) are dependent on the argument streams.

Deploying reactors on streams is very similar to calling lifted functions on behaviours in existing reactive languages. However, there are two important differences: First, whereas functions typically have only one return value, a reactor can declare *multiple* output expressions (in its `export` clause) that can be used, for example, as the input for deploying other reactors. Second, Haai reactors are entirely push-based. Most reactive languages actually use a combination of push and pull semantics: values are pushed through the dependency graph, but function calls happening during this process are treated in a pull-based way. For example, calling `+` in a traditional reactive language waits for the sum (i.e., pull) and subsequently pushes the sum through the rest of the DAG. `+` is a function and therefore does not know the notion of "dependent expressions". In Haai, even `+` is a reactor that has to be deployed and which follows the push-based philosophy. For instance, the expression `(+ x 5)` comprises four streams. The `+` is a built-in stream that perpetually (from a conceptual point of view) emits the `+` reactor. `x` is a stream (e.g., a formal parameter). `5` is a stream that, conceptually, emits the number 5. The entire expression is the stream created by deploying the `+` stream on the operand streams. The deployment pushes `x` and `5` into `+` and registers the "deploying reactor" (i.e., the site where the expression `(+ x 5)` occurs) as a dependency. Every time `+`, `x` or `5` emits a new value, the value of `x` and `5` will be pushed into the value of `+`. It is the implementation of `+` reactor that will trigger the "deployment-site" (rather than the "call-site" waiting for the result of the `+` function as is normally the case).

For all reactor deployments (both primitive reactors and user-defined reactors), reactors will start propagating values when all operand streams to a reactor deployment have consumed all their dependent values. If streams are producing values at different rates, the last value of every stream will be used by streams that are producing values slower compared with the other streams given to a deployment expression.

2.3 Pointwise Composition of Reactors

`define-reactor` syntax is used to define new reactors. The body of a reactor consists of a sequence of deployment expressions (some of which are given a local name with `define`). The last expression of a reactor is its `export` clause which lists one or more deployment expressions (typically

just local names). Each deployment expression defines a part of the functionality of the reactor. The order in which deployment expressions occur in a reactor is not taken into account for creating the dependency graph of the stream that emerges when deploying that reactor. A topological sort of all deployment expressions is performed at compile-time to avoid glitches and cyclic dependencies.

Since the body of a reactor consists of deployments of other reactors, Haai naturally supports pointwise composition in the same way that functional programming languages support pointwise composition of functions. An example of a reactor that illustrates this idea is shown in Listing 2. This reactor uses the existing `scale` and `rectangle` reactors by deploying them in the body of `rectangle-scale`. The input streams `w` and `h` are pushed into two instances of the `scale` reactor together with a scaling `factor`, and both outputs are then pushed into an instance of the earlier-defined `rectangle` reactor shown above. Notice that on Line 4 we use parentheses in `define` for defining multiple values that correspond to the deployment of `rectangle`. That is because `rectangle` has two outputs. One of them will push its emitted values to the dependencies on `a` and the other one will do the same for the dependencies on `p`.

Another alternative for composing reactors out of existing reactors (a point-free style) will be discussed in Section 3.2.

2.4 Stateful Reactors

Stateful reactors are reactors that have to (a) initialise some state as soon as they are deployed and (b) update that state for every incoming value emitted by those input streams. Consider for example the aforementioned `sum` reactor that has one input stream emitting numbers and which declares one output stream used to emit a new sum for every incoming number. Rather than trying to combine reactive programming (which is all about hiding evolving state) with imperative language constructs that initialise and assign local state variables, Haai provides programmers with a very disciplined way to describe changing state explicitly. `past` takes two arguments, namely a stream `s` and a default value `n`. `(past s n)` then corresponds to a stream that emits `n` the very first time `s` emits a value. After that (i.e. as soon as `s` has emitted at least one value) the previous value of `s` is emitted from `(past s n)`. `past` is inspired by ActiveSheet's [17] `PRE` operator and closely resembles Lucid Syncrone's [3] `pre` and `fby` operations.

Listing 3 shows the implementation of `sum` using `past`. The DAG that corresponds to `sum` is shown in Figure 2. The dotted line in the DAG, which is a back-edge in the graph, suggests a cyclic dependency. However, a cyclic resolution is avoided at run-time by the fact that the Haai compiler recognises the `past` special form by name: a hidden state variable is generated for `acc` and every time a new value for `acc` gets emitted, the old value is stored in that hidden state variable. The second argument of `past` corresponds to the initial value of that hidden state variable. In Figure 2, the state variable is depicted by a cloud. Every time a stateful reactor gets deployed, all the occurrences of `past` will first emit the initial value of that `past` expression. Like every other reactor, a stateful reactor can be deployed multiple times. All deployments of a stateful reactor manage their own state.

```
1 (define-reactor (sum a)
2   (define acc (+ a (past acc 0)))
3   (export acc))
```

Listing 3. Implementation of the `sum` reactor using `past`.

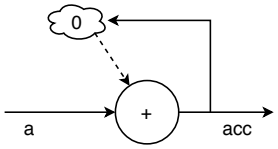


Figure 2. Directed acyclic graph showing data flow from the input to the output of the `sum` reactor.

3 First-Class Reactors

3.1 Streams That Emit Reactors

Streams are not first-class entities in Haai. Streams only emerge at the interpreter level whenever reactors are deployed. Every deployment results in one or more new streams (i.e., the `export` clause) that emit values as specified by the reactor. Whenever a deployment expression occurs in a `define`, the newly created stream is given a name that is local to the reactor and that can be referred to in other deployment expressions. Whenever a stream occurs in an export expression, it is not the stream that is exported but rather the values of the stream. In other words, the expression `(export a)` means that the reactor emits the values on stream `a`. Hence, streams are only "passed around" by the interpreter when deploying reactors. Streams are never emitted as values on streams.

Reactors, on the other hand, *are* first-class values. This means that a stream can emit reactors and that a reactor can be deployed with argument streams that emit reactors. Reactors whose arguments are streams emitting reactors or reactors that export streams emitting reactors are known as *higher-order reactors*. The difference between higher-order reactors and higher-order behaviours as found in other reactive languages is explained in Section 3.6.

In our explanation of deployment, we have explained the semantics of deployment expressions like `(r` r_1 `...` r_n`)`. In these expressions, all sub-expressions correspond to streams. The first sub-expression is expected to be a stream that is emitting reactors (just like the first sub-expression in

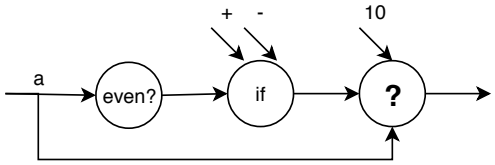


Figure 3. Dependency chain of a reactor stream deployment. At compile-time it is unknown which reactor is deployed on `a` and `10`.

a Scheme-expression `(e1 ... en)]` is expected to be a procedure at run-time). Otherwise, deployment makes no sense, and the error value is emitted from the stream that corresponds to the deployment expression. Since `r` is a stream of reactors, this means that deployment itself is a higher-order concept. This has two interesting consequences.

First, consider the following deployment expression:

```
((if (even? a) + -) a 10)
```

The stream corresponding to the innermost deployment expression can emit both the `+` reactor and the `-` reactor, but only one is used by the outermost deployment expression, depending on the value emitted on the `a` stream. A visualisation of the dependency graph is shown in Figure 3.

Second, the set of combinators that one typically finds in a reactive programming language (e.g., `map`, `filter`, ...) are higher-order reactors that can be implemented in Haai itself. For example, here is how the `filter` reactor can be implemented.

```
(define-reactor (filter r v)
  (export (if (r v) v)))
```

Remember from section 2.4 that the deployment of stateful reactors requires special attention. Consider the following higher-order deployment:

```
((if (> a 0) sum average) time)
```

Every time the sign of `a` changes, Haai will need to toggle between emitting the `sum` or the `average` for the deployment of the outermost deployment expression. This raises the question whether the second deployment of `sum` needs to proceed with the "old" state of its first deployment or whether this is considered a fresh deployment that needs to start from 0 again. In other words, it needs to choose between throwing away state when a different reactor is emitted by the higher-order stream or maintaining state for every encountered reactor. In Haai we opt for the first approach since the second one could potentially give rise to enormous amounts of deployments in case the sign of `a` changes frequently. In other words, every deployment (sub)expression in a Haai reactor gives rise to exactly one first-class reactor that is perpetually emitted by the stream that corresponds to that deployment (sub)expression. Hence, the above expression creates two deployments, one that corresponds to `(sum time)` and another one that corresponds to `(average time)`. The partial expression `(if (> a 0) sum average)` is to be seen as a stream that keeps on emitting one of the two existing reactors.

```
1 (define average
2   (ror (parallel* sum count) /))
```

Listing 4. Point-free composition of `sum` and `count` to implement `average`.

```
1 (define-reactor (average in)
2   (define s (sum in))
3   (define c (count in))
4   (export (/ s c)))
```

Listing 5. Pointwise composition of `sum` and `count` to implement `average`.

3.2 Point-Free Composition Operators

Haai supports point-free composition of reactors a result of reactors being first-class values. Three built-in composition operators have been defined which compose two constituent reactors into a new composite reactor. These operations are `ror` ("reactor after reactor"), `parallel` and `parallel*`. `ror` creates new composite reactors that are created by sequential composition of its argument reactors. `parallel` and `parallel*` both create new composite reactors that are created by parallel composition of its argument reactors. `parallel*` passes all inputs to the composite reactor to both constituent reactors, and `parallel` will pass the first *n* inputs to the first constituent reactor, when the first constituent reactor has *n* inputs, and the remaining inputs to the second constituent reactor. These operations are implemented as first-class higher-order reactors that take two reactor streams as operands, and emits the composite reactors on its output stream.

Listing 4 exemplifies the usage of `ror` and `parallel*`. Given a reactor `sum` that calculates the sum of its input values and a reactor `count` that counts the number of input values (`sum` and `count` are stateful reactors, see Section 2.4). The `average` reactor combines these two reactors to calculate the average of all values emitted by its input stream. The corresponding DAG is shown in Figure 4. Notice that this reactor is fully equivalent to the reactor definition shown in Listing 5 which uses pointwise composition.

3.3 `rho`: The `lambda` of Reactive Programming

In functional programming, higher-order functions go hand in hand with anonymous functions. For example, in Scheme, incrementing all the elements of a list `l` is typically done using the expression:

```
(map (lambda (x) (+ x 1)) l)
```

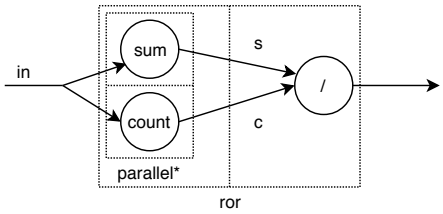


Figure 4. Directed acyclic graph showing the data flow from the input to the output of the `average` reactor.

```
(define f-to-c
  (ror (rho (f) (export (- f 32)))
       (rho (a) (export (* a 0.5556)))))
```

Listing 6. Example of how `rho` can be used for point-free composition using `ror`.

```
(define-reactor (creator num-clicks)
  (export (rho (t)
            (export (if (even? t)
                        (/ t 2)
                        (+ (* t 3) 1))))))
```

Listing 7. Reactor that exports an anonymous reactor that can be deployed elsewhere.

Lambdas are not only useful for calling higher-order functions but also to pass a function defined in the same scope to another function.

Haai adapts these ideas to the realm of reactive programming. In Haai, anonymous reactors can be created using the `rho` construction. The syntax of `rho` is identical to that of `define-reactor`, except that the parameter list is not prefixed with a name for the reactor. Listing 6 shows the point-free composition of two anonymous reactors with the `ror` operator discussed in the previous section. The first anonymous reactor reacts by subtracting `32` form the values emitted by `f` and the second anonymous reactor reacts to incoming values by multiplying them by `0.5556`. Thus converting temperatures in Fahrenheit to Celsius.

Apart from using anonymous reactors as parameters of a higher-order reactor, they can also be emitted from a reactor. Listing 7 shows a reactor called `creator` that reacts to one input stream `num-clicks`. Every time the latter emits a value, the stream created by deploying `creator` will emit a first-class reactor. Notice that the same reactor is emitted time and time again (conceptually) since `num-clicks` is not used in the `rho` expression.

Anonymous reactors can be used to implement simplified versions of the composition reactors as long as the number of input streams is static. Reactors, unlike procedures in Scheme, do not allow an arbitrary amount of input streams[4],

[4] With the exception of the built-in arithmetic reactors.

```
(define-reactor (creator num-clicks)
  (export (rho (a)
            (export (if (even? a)
                        num-clicks)))))
```

Listing 8. Reactor that exports an anonymous function that can be deployed elsewhere. Due to scoping the stream `num-clicks` is also used in the anonymous reactor. And all deployments of the anonymous reactor add an implicit dependency on this stream.

as the existence of such a construct would result in either a variable not being a stream, or result in a stream that contains a list of streams, which is not possible since streams are not first-class entities.

3.4 Captures

Anonymous reactors can occur inside the body of other reactors, and therefore can be seen as **nested reactors**. Anonymous reactors are subject to lexical scoping. Consider the example in Listing 8 which is a modification of the `creator` reactor from Listing 7. In this example, a stream that was created by deploying `creator` will emit a new `rho`. However, in this case, the `num-clicks` that is in the lexical scope of the anonymous reactor is effectively used in the body of the `rho`. Hence, any stream that is created by deploying that `rho` will have to emit those values emitted on `num-clicks` at the time when an even value is emitted by the input stream `a`. Hence the anonymous reactor that is emitted by the stream corresponding to `creator` needs to maintain a reference to all the streams that are in the surrounding lexical scope of an anonymous reactor. During compilation, it is possible to determine which streams from the surrounding lexical scope are used by nested anonymous reactors.

Analogous to closures in functional languages, first class reactors are represented by so-called *captures*. A capture is the run-time value that corresponds to a `rho` expression just like a closure is the run-time value that corresponds to a `lambda` expression. Conceptually, inside the reactive interpreter, a capture for a `rho` is a couple that consists of a DAG that corresponds to the body of the `rho` and a list of all the streams which the `rho` inherits from its surrounding lexical scope. In the `rho` shown in Listing 8, the `creator` reactor thus conceptually emits a stream of captures. All those captures consist of the DAG that corresponds to the body of the `rho` and a reference to the `num-clicks` stream.

3.5 Deploying Captures

Now we that we know that first-class reactors are actually captures emitted by a stream, we can think of deploying those captures. This is needed for deployment expressions where the deployed reactor corresponds to a stream of captures which are not all the same. Consider the example shown

```
1 (define-reactor (temp-conversion temp
      in-celsius)
2   (export
3     (if in-celsius
4         (rho (out-celsius)
5           (export (if out-celsius temp (* (-
                temp 32) 0.5556))))
6         (rho (out-celsius)
7           (export (if out-celsius (+ (/ temp
                0.5556) 32) temp))))))
```

Listing 9. Temperature conversion between fahrenheit and celsius, and vice versa, by making use of anonymous reactors.

in Listing 9. Deploying this reactor with a given temperature stream `t` and a stream of booleans `b` with the expression `(temp-conversion t b)` results in a stream that perpetually emits one of both `rho` captures from the body of `temp-conversion` based on the boolean value that is emitted by `in-celsius`.

Deploying a capture means that the given input streams (i.e., the deployment arguments), as well as the implicit streams stored in the capture need to be connected to the body DAG of the capture. When captures in a deployment expression are emitted by a higher-order reactor, then the input streams need to be disconnected from the previous capture emitted by the higher-order reactor and reconnected to the next capture emitted by the higher-order reactor. Caution is required, as naively disconnecting them can lead to subtle errors. When a reactor that emits captures using `rho` is (temporarily) no longer deployed, its previously emitted captures may still be deployed elsewhere in the program. Thus, it is possible that, although the `temp-conversion` reactor is no longer deployed, that it is still active indirectly via the deployment of one of its once-emitted captures elsewhere in the program. If the captures would have a dependency on a stream created in `temp-conversion` (e.g. the incoming temperature is first converted to Kelvin), then this stream should still be listening to the incoming events passed on the `temp` stream, even when the `temp-conversion` reactor is not directly deployed.

Alternative approaches were considered for anonymous reactors. In one alternative approach, *snapshots* would be used instead of captures. In this approach, the last value of each dependent stream would be stored. However, this was deemed less expressive as a deployed capture can listen to a stream found in the lexical scope, even if the reactor were the capture was created is no longer deployed, which is not possible using snapshots.

3.6 Higher-Order Reactors vs. Behaviours

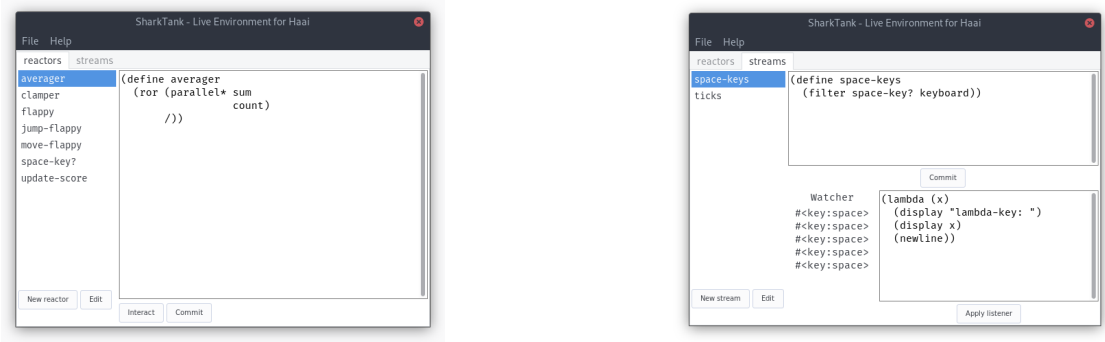
Haai features higher-order reactors, which is different from the traditional higher-order signals found in other reactive programming languages. Examples of signals of signals, or streams of streams, can be found in FlapJax [15], and Reactive Extensions [2]. We argue that this is a problematic feature that is very difficult to grasp since it requires a programmer to understand two-dimensional time: the outer stream is emitting streams which are at the same time all emitting values. This is avoided by disallowing such streams, and introducing first-class reactors. Deploying a higher-order reactor may result in a stream that emits reactors. However, these are "dead" reactors that have not been deployed yet. This is much easier to understand. These "dead" reactors (or more precisely: captures) only start "living" when they are deployed because that is the moment that their DAG gets connected to input streams. In summary, we conjecture that it is easier to understand a reactor that emits deployable reactors than a reactor that emits reacting reactors.

4 Live Environment

Live Programming aids program development by blurring the distinction between editing code and running the program. Thus developers do not have to mentally switch between "developing" and "testing". It minimises the effects of the "feedback loop" that occurs whenever a programmer tries to verify the effects of changing the code of the program, however minimal or large a change may be.

A live programming environment for a reactive programming language needs to define the *granularity of code changes* that can be applied while the program is running, i.e., it needs to define what exactly constitutes one "edit step". Live programming environments for reactive programming languages currently use one of two approaches. The first approach is used by an experimental time-travelling debugger for Elm [5] where the granularity of change is nothing less than a recompilation of the entire program. Unfortunately, in the context of live programming reloading the entire program is a costly operation that should be avoided. The second approach, used by Glitch [13], conceptually reloads the program on every individual keystroke. Keystrokes that leave the program in a broken state (e.g., parser or compiler errors) are not visualised in the programming environment. We believe that reusing reactors as a linguistic element that defines the granularity of change is a sweet spot between these two radical extremes. This approach is similar to how objects and classes define the granularity of change in Self and in SmallTalk live programming environments.

In previous sections, we discussed that a reactive program written in Haai is one giant reactor that typically consists of multiple smaller constituent reactors, and that the constituent reactors can be changed without touching the reactor(s) that use it in some (point-free or pointwise) composition. In essence, modifying a reactor amounts to installing a new version of said reactor, which automatically causes any dependencies on the old constituent reactor to be replaced



(a) View to manage reactors.

(b) View to manage streams.

Figure 5. Overview of the SharkTank Live Reactive Programming Environment.

with dependencies to the newly installed constituent reactor. Based on these principles we introduce *SharkTank*, a prototypical live programming environment for the Haai reactive programming language.

4.1 SharkTank in Action

Figure 5 shows the SharkTank editor. When the editor is started, two views can be used to inspect the running reactive program. The view in Figure 5a gives an overview of all user-defined (named) reactors in the running program. The pane on the left shows a list of those reactors, and the text field on the right shows the source code of the currently selected reactor in that list. Anonymous reactors are omitted from the list since they can only be modified by modifying their encapsulating reactor. When the source code of a reactor is modified, it can be committed by clicking the "Commit" button. This action will cause all deployments of that particular reactor to be replaced by deployments of the newly committed reactor. It can be the case that, according to the updated behaviour of a named reactor, (a) modified anonymous capture(s) has to be emitted. Such updated captures are propagated throughout the program as soon as the named reactor that produces them gets (re)deployed. Apart from the "Commit" button, the view in Figure 5a also shows an "Interact" button that leads the programmer to a facility to test reactors offline, a feature that is outside the scope of this paper.

The view in Figure 5b is used manipulate streams which correspond to the deployment of some of the reactors that make up the reactive program (more precisely: the streams that have been explicitly created by the programmer; not the streams that correspond to deployments in reactor expressions). This view gives an overview of all the streams in the current application that were manually created by the programmer (the leftmost pane), the source code of the currently selected stream (the topmost text field), a "watcher" that lists the 5 most recent values emitted by the selected stream (middle-left), and finally, a text field to attach a Scheme "listener" to the selected stream (explained further below). Similarly to editing the source code of a reactor, the definition of a stream can be edited while the program is running. In this case, an implementation is shown of the `space-keys` stream that filters the built-in stream of all keystrokes, such that it only contains events of the `space` key being pressed.

Conceptually the SharkTank live programming environment can be seen as a layer built on top of the Haai runtime which is itself written in an imperative language and which is connected to the "imperative world" (e.g. storage, input-output, ...). Even in order to just "see the output" of a reactive program, a link needs to be established between some of the streams and this imperative world: even a simple display of the results emitted by the program corresponds to an imperative side-effect on the screen. The problem of communicating results from a reactive program to an imperative one has been previously discussed in [9]. SharkTank allows a programmer to establish this connection by attaching a Scheme "listener" to some of the streams. These Scheme listeners are 1-argument callbacks that will be called every time a value is emitted on the stream. In future versions of SharkTank we plan to use the actor-reactor model [7] to cleanly separate the threads that manage reacting streams from the threads that manage the imperative world.

4.2 Updating State

Reactors that use `past` for their internal definitions are stateful, and these reactors may be redefined via SharkTank as well. Ideally, the internal state of a reactor is preserved when a new version of that reactor is committed by the user. However, since any changes to reactors are permitted, often the old state will not be compatible with the new structure of the reactor, or if the structure is similar enough the semantics of the state may have changed. Therefore the internal state created in all previous deployments are completely discarded when a new version of that reactor is installed by the live programming environment. In future versions of SharkTank we may explore strategies to keep internal state, but other

than replaying all events since the start of the program we do not know of a general solution to this problem.

5 Related Work

In this section, we discuss the reactive languages, libraries, and live programming environments that are, from our point of view, most related to the goals of Haai.

FrTime [4] is a functional reactive programming language built on top of Racket. While FrTime has been a very influential language in the field of reactive programming with respect to the representation of signals and glitch prevention, it does not facilitate higher-order reactive programming.

Flapjax [10] is a reactive programming language for building interactive web applications in JavaScript that is inspired by FrTime. One of its most interesting features related to Haai is support for higher-order reactive programming via nested event streams. However, besides a limited number of built-in operators that can be applied to higher-order streams, Flapjax does not define the linguistic concepts that are necessary to handle the complexity of higher-order reactive programs. We believe one of the main differences between higher-order reactive programming in Haai and Flapjax is that Haai adds an extra layer which is the concept of reactor deployment[5]. When a reactor is produced as the result of some computation, it can only become active once it is deployed by another reactor. In contrast, event streams in Flapjax can produce other event streams as their values, and conceptually all of those inner event streams immediately start producing events when they are conceived, regardless of whether they are used in the rest of the program or not.

Lucid Synchrone [3] is a functional synchronous programming language that closely resembles Haai. It has support for higher-order functions to be passed around, but streams of functions are not automatically used to reconfigure the program, which is a manual operation compared to the dynamic deployment of streams of reactors in Haai. Furthermore, Lucid Synchrone contains the notion of clocks to support inputs that have a different production rate. This is absent in Haai since reactors will react on any change, at any moment in time.

Elm [6] is a purely functional reactive programming language for building interactive web applications. During the early days of its research, there has been some experimentation using Elm for building a time-travelling debugger to debug and modify reactive programs while they are running [5]. The granularity of an "edit step" in this time-travelling debugger is the recompilation of the entire reactive program. In contrast, the granularity of program modifications in Haai is on the level of individual reactors rather than a completely new deployment of the program wherein state can be (possibly) restored.

[5] Nested event streams as they occur in languages like Flapjax are still available in Haai by deploying a reactor that produces the nested stream.

SuperGlue [12, 14] is a live-textual language inspired by declarative data-flow visual languages. It allows developers to program interactive applications as a combination of objects and signals. Although SuperGlue's signals facilitate the construction of a reactive data-flow model to support "liveness", the objects themselves are not reactive, and the abstractions for representing signals do not integrate with the rest of the language.

A prototype by Schuster and Flanagan [16] uses a combination of a textual and visual live programming environment for building web applications. In this environment, programmers can change string literals in the source code of a non-reactive language (JavaScript). Furthermore, it allows developers to replay execution of the program via direct manipulation within a graphical user interface. The prototype does not facilitate live editing of reactive programs.

ZenSheet [8] is a programming environment for reactive computations that generalises the concept of a spreadsheet to provide developers with an environment that combines textual and visual live programming features. Its language, Peano, uses "lazy variables" as the backbone of its reactive model. Contrary to Haai, Peano allows developers to write imperative expressions, and it does not support higher-order reactive abstractions.

ActiveSheets [17] is a stream programming platform for spreadsheets. Its novel features are importing of live data in spreadsheet cells, a windowing mechanism to segment live streams into static ranges of data over time, and the introduction of stateful cells. Since spreadsheets typically do not allow any form of recursion or side-effects in their formulas, every spreadsheet program can be seen as a large reactor.

6 Limitations and Future Work

Further research can focus on incorporating the notion of Edit Transactions [11] into Haai. Currently, one "commit" can only modify the implementation of one reactor at a time. If a change to a reactor requires other reactors to be modified as well (e.g., the number of input or outputs streams of a reactor changes), then changing every reactor one-at-a-time will result in errors. Edit Transactions can be useful to group multiple changes into one change that is applied as a single transaction.

Furthermore, Haai has some more fundamental restrictions. There is no support for recursion, iteration nor side-effects by design to ensure strong reactivity [7]. Furthermore, there can be no cycles in the dependency chain, except where they can be fixed by making use of `past` to refer to a previous value emitted on a stream. A less fundamental limitation, is the absence of compound data types (such as pairs, vectors, dictionaries,...). This restriction has been set in place to focus first on the semantics of higher-order reactive programming by means of first-class reactors. We expect that

constructors and accessors, and even functional mutators, can be implemented as reactors.

7 Conclusions

In this paper we introduced Haai, a reactive programming language for higher-order reactive programming. At the basis of Haai is the concept of the reactor that serves as an abstraction over a reactive program. Reactor composition lies at the basis of Haai programs, which can be done in both a pointwise and point-free style.

With Haai we investigated a new way to construct higher-order reactive programs via first-class reactors. One of the key differences in relation to previous approaches for writing higher-order reactive programs is the concept of "reactor deployment". Whereas in previous approaches a reactive value is usually "active" from its conception, reactors only become active once they are deployed. We believe this is an important step to precisely define higher-order reactive programming with clear and concise semantics.

We used Haai to build SharkTank, a prototype live programming environment for reactive programming languages. Whereas previous approaches required either a full-program recompilation or modified the running program on a per-keystroke basis, in SharkTank unit of "change" is on the level of individual reactors. Because Haai is reactive all the way down, it is possible for the live programming environment to deploy new versions of existing reactors, which will automatically propagate throughout the program like any other reactive value. As only the deployments of the modified reactor need to be updated, the impact of a live change is more manageable compared to previous approaches.

Acknowledgments

Sam Van den Vonder is funded by the Research Foundation - Flanders (FWO) under grant number 1S95318N. Humberto Rodriguez Avila is also supported by the FWO under grant number 8B02.

References

[1] 2018. Light Table. http://web.archive.org/web/20180830122311/http://lighttable.com/. (2018). Accessed: 2018-08-30.

[2] 2018. ReactiveX: An API for asynchronous programming with observable streams. https://web.archive.org/web/20180829121631/http://reactivex.io/. (2018). Accessed: 2018-08-29.

[3] 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. DOI: http://dx.doi.org/10.1145/1176887.1176899

[4] 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)*, Peter Sestoft (Ed.), Vol. 3924. Springer, 294–308.

[5] Evan Czaplicki. 2013. Interactive Programming: Hot-swapping in Elm. https://web.archive.org/web/20180831090231/http://elm-lang.org/blog/interactive-programming. (2013). Accessed: 2018-08-31.

[6] 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.

[7] Sam Van den Vonder, Joeri De Koster, Florian Myter, and Wolfgang De Meuter. 2017. Tackling the awkward squad for reactive programming: the actor-reactor model. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, Vancouver, BC, Canada, October 23, 2017*, Guido Salvaneschi, Wolfgang De Meuter, Patrick Eugster, and Lukasz Ziarek (Eds.). ACM, 27–33.

[8] Monica Figuera. 2017. ZenSheet studio: a spreadsheet-inspired environment for reactive computing. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, SPLASH 2017, Vancouver, BC, Canada, October 23 - 27, 2017*, Gail C. Murphy (Ed.). ACM, 33–35.

[9] 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)*, Masami Hagiya and Philip Wadler (Eds.), Vol. 3945. Springer, 259–276.

[10] John H. Maloney and Randall B. Smith. 1995. Directness and Liveness in the Morphic User Interface Construction Environment. In *Proceedings of the 8th Annual ACM Symposium on User Interface Software and Technology, UIST 1995, Pittsburgh, PA, USA, November 14-17, 1995*, George G. Robertson (Ed.). ACM, 21–28.

[11] Toni Mattis, Patrick Rein, and Robert Hirschfeld. 2017. Edit Transactions: Dynamically Scoped Change Sets for Controlled Updates in Live Programming. *Programming Journal* 1, 2 (2017), 13.

[12] Sean McDirmid. 2007. Living it up with a live programming language. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. (Eds.). ACM, 623–638.

[13] Sean McDirmid. 2013. Glitch: A Live Programming Model. (2013).

[14] Sean McDirmid and Wilson C. Hsieh. 2006. SuperGlue: Component Programming with Object-Oriented Signals. In *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings (Lecture Notes in Computer Science)*, Dave Thomas (Ed.), Vol. 4067. Springer, 206–229.

[15] 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.

[16] Christopher Schuster and Cormac Flanagan. 2016. Live Programming by Example: Using Direct Manipulation for Live Program Synthesis. In *LIVE Workshop*.

[17] 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)*, Richard E. Jones (Ed.), Vol. 8586. Springer, 360–384.