A Graph-Based Formal Semantics of Reactive Programming from First Principles

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

Joeri De Koster
joeri.de.koster@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

In recent years, stream processing has become the de facto paradigm to process any kind of real-time data in many kinds of applications. Different libraries, frameworks and techniques exists which aim to make it easy to build stream processing applications in many modern programming languages…Libraries such as Reactive Extensions, Akka Streams, or web frameworks such as React and Vue are all based on the idea of data streams that are connected to graphs to model the flow of data in applications. To the best of our knowledge, there exist no formalism which captures the essential core semantics of these approaches in a straightforward, easy to understand, manner: namely its graph-based program structure and the turn-based propagations of values through this graph. In this paper, we present *Karcharias*, a formalisation of reactive programming (a paradigm that shares many core ideas found in the various aforementioned libraries and frameworks) that is built from first principles. Instead of extending an existing language with a graph-based stream processing framework, and formalising this integrated language, we formalised the reactive programming paradigm without relying on a base language (e.g., the λ-calculus). Using our formalism, we show how reactive programs (and thus, stream-based programs in general) need a way to construct a graph and to propagate events through that graph, even in the absence of a base language.

CCS CONCEPTS

• **Software and its engineering → Data flow languages**; • **Theory of computation → Operational semantics**.

KEYWORDS

Reactive Programming, Operational Semantics

ACM Reference Format:
Bjarno Oeyen, Joeri De Koster, and Wolfgang De Meuter. 2018. A Graph-Based Formal Semantics of Reactive Programming from First Principles. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 8 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Stream processing has become a popular paradigm to process data in various kinds of applications. One approach of stream processing that has gained popularity in recent years is called reactive programming (RP) [2, 11]. Reactive programs declare the dependencies (i.e. constraints) between the time-varying signals that make up the program. Whenever a signal updates its value, all signals that depend on that signal are updated by recomputing their values, according to the specified constraints.

In this paper, we make a distinction between two different implementation styles (*strains*) for reactive programming languages (RPLs). *Function-based reactive programming languages* usually model reactive programs using signal functions[1]. At each turn (i.e. change of an external, with respect to the RP program, input source that causes a re-computation), a value is propagated through a signal function (which may be composed out of different, smaller, signal functions) to not only produce an output, but also the signal function to use in the next turn. Examples of languages that implement function-based RP with signal functions, and variations thereof, are Yampa [24], SFRP [4], and Dunai [27].

Graph-based reactive programming languages model reactive programs as graphs. During the evaluation of graph-based reactive programs, a graph data structure is constructed (often refered to as the *dependency graph*) where nodes correspond with the signals and (directed) edges between the nodes correspond with the data dependencies between these edges. Instead of applying functions that produce values and (updated) signal functions, turns in graph-based RP are performed by propagating values along the edges of the dependency graph. A node (signal) is updated when one of its dependent nodes changes. This update is performed once all the dependent nodes have had a chance to update (to ensure the absence of glitches [5]), e.g., by performing the updates in the dependency graph in topological order (an approach often used when the dependency graph is static) or by using a height-ordered priority queue. Examples of RPLs that implement graph-based RP are FrTime [5], Frappé [6], REScala [28], and Flapjax [23].

Roughly speaking, each strain finds its origin in a different research community. Function-based RPLs find their origins in the world of functional programming (and are often implemented in functional programming languages like Haskell and Agda). They encourage programmers to compose programs in a pointfree style using operators like >>>, &&& and *** [4], often by making use

[1]Before signal functions, many function-based RPLs modeled signals as pure functions that return, given a timestamp [11] (or a list of timestamps, e.g., as in [34]), the corresponding value(s) of a signal. We ignore their existence in the rest of the paper since they have been mostly superseded by signal functions [20].

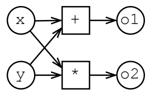

Figure 1: Simple graph-based reactive program visualised as a Directed Acyclic Graph (DAG).

of arrows [14]. Research has been focused on supporting higher-order reconfigurations of RP programs (via so-called `switch` operators, of which there usually exist multiple with subtle differences in semantics), and using type systems to statically verify (e.g.,) liveness [1, 30]. On the other hand, graph-based RPLs often find their origins in object-oriented programming (e.g., in Scala [21]). Research on graph-based RPLs has been focused on efficiency (e.g., Emfrp [29], REScala [10], and ReactiFi [32]), integrating reactive code with imperative code (e.g., FrTime [15], and Stella [9]) and making RP work for distributed applications (e.g., XFRP [31], and AmbientTalk/R [3]). Of course, there exists research that aims to enhance function-based RPLs on similar fronts, although from a much more theoretical perspective [19, 35, 36].

In this paper, we aim to formalise graph-based reactive programming from a first principle approach. We do this by formalising Haai, a paradigmatically pure graph-based RPL. Unlike many other RPLs, Haai programs are constructed without relying on non-reactive base language. In other words, Haai does not have functions, only reactors (first-class graphs that describe (a part of) a reactive program). Therefore, Haai lacks the notion of lifting.

Our formalisation of Haai, which we have named *Karcharias*, provides an intuitive understanding of graph-based RP. The small-step semantics provide a clear formalisation of how RP programs operate over time (e.g., by recomputing the program with respect to some *current* values). By formalising reactive programs as graphs where nodes have values that change over time, it is easier to reason about the memory allocation and consumption behaviour of RP programs, compared to most formalisations of function-based RPLs [16–18] which often relies on recursion and various memory-intensive operations to model the time-varying nature of signals.

2 A BRIEF INTRODUCTION TO HAAI

We first present a brief informal overview of Haai, the reactive programing language which we have based our formalisation on. Inspired by Lisp and Scheme, Haai uses s-expression syntax to denote *reactors*. Reactors are the core abstraction of Haai, they describe the structure of a reactive program. For example, the reactive program

```
(defr (sum-and-product x y)
  (out (+ x y)
       (* x y)))
```

defines a reactor (as a DAG, see Figure 1) that computes both the sum and product of two numbers and "returns" both of them (`out` is a special syntactic construct used to define multiple sink nodes; it is similar to `values` in Scheme). Instead of applying a procedure or a function as in non-reactive languages, reactors in Haai are *deployed* (i.e. instantiated) on signals whose value may change over time. We call an instantiation of reactor a *deployment*. For example, assuming that there are two signals `time` and `velocity`, the `sum-and-product` reactor can be deployed as (`sum-and-product time velocity`). This produces two new signals, containing respectively the sum and product of the two inputs, which will be updated whenever either `time` or `velocity` (or both) changes. The exact approach taken to update signals in an efficient manner is out of the scope of this paper.

Haai is a higher-order reactive language which allows dynamic reconfigurations of the dependency graph, without the need for carefully-designed switching operators. For example, in

```
(defr (twice r x)
  (r (r x)))
```

a higher-order reactor is defined which deploys a reactor given as input twice, connecting the sinks of one deployment to the sources of the second. As reactors are first-class values, a signal can carry them as their current value. For example, the expression `(if (even? time) + -)` corresponds with a signal whose value is either the `+` reactor or the `-` reactor, depending on whether `time` is even or odd. In other words, deployments of higher-order reactors contain holes which, at run-time, are filled in with the deployment of the reactor carried by the operator signal.

Deployments in Haai are disabled if they are not in use, which happens if the operator signal carries a different reactor w.r.t. a previous turn. Deployments are thus re-enabled when the operator signal carries the same reactor again. We call this kind of semantics *toggle semantics*.

Finally, Haai has support for anonymous reactors. For example,

```
(defr (make-twice r)
  (rho (x)
    (r (r x))))
```

uses the `rho` syntactic form to create an anymous reactor. Reactors employ lexical scoping. In other words, the two occurrences of `r` in `(r (r x))` refer to the signal node `r` as defined in the first line. When the anonymous reactor is deployed, both `r`s refer to the `r` signal from its lexical scope. We call these reactors with lexical scope *captures* as they capture the signals from their environment (captures are similar to closures in non-reactive languages).

This concludes our brief overview of Haai. We refer to earlier work on Haai for more details about the language: e.g., the ability to recursively generate reactive programs is discussed in [25] and state management with deployments is discussed in [26].

3 OPERATIONAL SEMANTICS

This section presents the small-step operational semantics of graph-based reactive programming. An implementation of these semantics, using PLT Redex [12], can be found online[2].

3.1 Syntax

Figure 2 presents an overview of the syntax of Karcharias. In Karcharias, a reactive program (p) consists of a set of reactor definitions. Each reactor definition gives a name to a graph. In graph-based reactive programs, the order of the connections in the graph matters. Inspired by A-Normal Form [13], we encode reactors (i.e.

[2]https://gitlab.soft.vub.ac.be/boeyen/karcharias/

$$\begin{array}{rcl}
p \in \textbf{Program} & ::= & R \\
r \in R \subseteq \textbf{Reactor} & ::= & \mathcal{R}\langle x, N\rangle \\
n \in N \subseteq \textbf{Node} & ::= & (\bar{i}, nt, \bar{o}) \\
i \in \textbf{Input Port} & ::= & x \mid v \\
nt \in \textbf{Node Type} & ::= & \mathscr{RHO}\langle N\rangle \\
 & \mid & \mathscr{DEPLOY} \\
o \in \textbf{Output Port} & ::= & x \\
\end{array}$$

$$x \in X \subseteq \textbf{Name}$$
$$\{in_{i,j}, out_{i,j} \mid \forall i \in \mathbb{N}^+, \forall j \in \mathbb{N}\} \subseteq X$$
$$v \in V \subseteq \textbf{Domain Value}$$

Figure 2: Syntax of Karcharias.

DAGs) as a set of nodes (N) where each node is a triple containing the inputs (either names refering to signals defined in the lexical environment or defined locally by another node, or constant values that are $\in V$[3]), the type of the node nt, and the outputs (which are the names to define new signals with).

The reactive program in Figure 1 can be modelled as only two nodes: one for each deployment expression $(([+, in_{1,0}, in_{2,0}], \mathscr{DEPLOY}, [out_{1,0}])$ and $([/, in_{1,0}, in_{2,0}], \mathscr{DEPLOY}, [out_{2,0}]))$. The names $in_{i,j}$ and $out_{i,j}$ are special as they denote the input and output signals of reactors: the first index denotes the index of the source or sink signal (1-based indexing), and the second index denotes the scoping level (similar to De Bruijn Indices which were originally invented for λ-calculi [8]; using 0-based indexing) in case of nested reactor definitions (without nested reactor definitions or lexical scoping there is no need for the second index). Note that the operator itself is one of the inputs of the $\mathscr{DEPLOY}$, in order to support dynamic reactive programs (programs whose dependency graph can change at run-time).

There are two supported node types (nt) in Karcharias. The aforementioned $\mathscr{DEPLOY}$ nodes denote deployments of DAGs (i.e. holes in one graph that have to be filled in, at run-time, by another DAG). $\mathscr{RHO}\langle N\rangle$ nodes denotes in-line DAG (reactor) definitions that have access to their lexical scope. I.e. a signal defined in the right-hand side of a node n is also accessible by the $\mathscr{RHO}\langle N\rangle$s of the same reactor. To disambiguate between the source signals of the nested and the surrounding DAGs, the source signal accessible as $in_{1,0}$ in the surrounding DAG is accessible as $in_{1,1}$ in the nested DAG. The same applies to sink signals ($out_{1,0}$ becomes $out_{1,1}$ in the nested DAG).

In the rest of the paper, we assume that programs are well-formed: we assume that the inputs and outputs are correct for every node[4], that reactors are acyclic, that there are no free variables (except for globally-defined signals and operators such as `time` and `+`, which are discussed later), and that each reactor has at least one sink node (i.e. $out_{1,0}$ has to be defined in every $\mathcal{R}\langle x, N\rangle$).

3.2 Semantic Entities

Reactive programs in Karcharias are executed in turns. In each turn, the values of a certain set of output signals are computed in terms of the values of certain external time-varying signals (such as `time`). We first explain the intra-turn semantics in Sections 3.2 to 3.4, and later use the intra-turn semantics to define in the inter-turn semantics in Section 3.5.

$$\begin{array}{rcl}
k \in K \subseteq \textbf{Configuration} & ::= & \mathcal{K}\langle E, I_d, W, S, D\rangle \\
w \in W \subseteq \textbf{Deployment Wiring} & ::= & \mathcal{W}\langle \iota_d, N, \Sigma\rangle \\
s \in S \subseteq \textbf{Deployment Snapshot} & ::= & \mathcal{S}\langle \iota_d, \Sigma\rangle \\
c \in \textbf{Capture} & ::= & \mathcal{C}\langle \iota_c, N, \Sigma\rangle \\
\sigma \subseteq \textbf{Signal} & ::= & v \\
 & \mid & \mathcal{S}_{\mathscr{GLB}}\langle x\rangle \\
 & \mid & \mathcal{S}_{\mathscr{REF}}\langle \sigma, x\rangle \\
 & \mid & \mathcal{S}_{\mathscr{DEP}}\langle \iota_b, \sigma, \overline{\sigma}\rangle \\
i & ::= & \ldots \mid \sigma \\
v & ::= & \ldots \mid p \mid c \mid \iota_d \mid \overline{v} \\
E \subseteq \Sigma \subseteq \textbf{Value Environment} & ::= & \{x \mapsto v, \ldots\} \\
\Sigma \subseteq \textbf{Signal Environment} & ::= & \{x \mapsto \sigma, \ldots\} \\
D \subseteq \textbf{Toggle Environment} & ::= & \{(\iota_b, \iota_c) \mapsto \iota_d, \ldots\} \\
\end{array}$$

$$\iota_b \in I_b \subseteq \textbf{Branching Point Id}, \iota_c \in I_c \subseteq \textbf{Capture Id}, \iota_d \in I_d \subseteq \textbf{Deployment Id}$$
$$p \in \textbf{Primitives}, \delta_p : V* \to V* \text{ (for every } p)$$

Figure 3: The semantic entities of Karcharias used in the operational semantics.

The semantic entities needed to express the intra-turn semantics are shown in Figure 3. Remember that, at run-time, a reactive program (i.e. a **capture**) is instantiated into a number of **deployments**, which has (usually) created new **signals**. These signals can then produce values in every turn. We describe these semantic entities in order, before discussing the **configurations** and the **primitive operations**.

3.2.1 Captures. Captures are represented as a tuple containing a unique identifier ι_c, a set of nodes N, and a lexical environment Σ, which contains the bindings for the captures signals. Named reactors (those defined in p) have, among other entities, the external signals (such as `time`) in scope. We will explain this in further detail in Section 3.3.

3.2.2 Deployments. Deployments are represented by two seperate entities: a deployment wiring, and a deployment snapshot. Information about the structure of the reactive program’s dependency graph is maintained in a deployment wiring. Each wiring consist of an unique identifier ι_d (which identifies the deployment), a set of uninstantiated nodes N and a signal environment Σ. Information about the current values of the signals of a deployment are stored in a deployment snapshot. Each snapshot consist of a unique identifier ι_d (which is always shared with a deployment wiring) and a signal environment Σ. In Section 3.4, we will describe the reduction rules that operate on these wirings (i.e. to go from a set of nodes N to a signal environment Σ) and snapshots (i.e. to convert the signal environment Σ into a value environment E). The main idea here is that the deployment wirings can be retained across turns as they represent the structure of the dependency graph, independent of the values of the time-varying values.

When the program is running, the $\mathscr{DEPLOY}$ nodes will need to be replaced with instantiated deployments. To avoid needlessly re-creating parts of the dependency graph, it is important that the deployments that have filled in the holes introduced $\mathscr{DEPLOY}$ are kept in-between turns. Thus, every instantiated deployment node is represented by a branching points from I_b, and all information regarding the location is stored in D (a toggle environment).

3.2.3 Signals. There are four types of signals in Karcharias. 1) v signals are constant signals whose value never changes over time, which does not only include the domain values from Section 3.1,

[3]Our formalisation is agnostic to the types of the domain (base) values. One can think of V as being the set of numbers, booleans, strings…

[4]At least one input (the operator) and one output for each $\mathscr{DEPLOY}$. All captured signals of N in $\mathscr{RHO}$ are present in the inputs of the nodes, and exactly one output (of which the capture will be stored).

but also primitive operation objects (see Section 3.2.5), captures, deployment identifiers (ι_d) as well as sequences of values (which is used to represent the sink values of any of the primitive operations). 2) $\mathcal{S}_{\mathcal{GLB}}\langle x\rangle$ signals are references to external signals, whose current value is stored in the configuration. 3) $\mathcal{S}_{\mathcal{REF}}\langle\sigma, x\rangle$ references a signal with a particular name (second element) as stored in a deployment's (first element[5]) signal environment. 4) And finally, $\mathcal{S}_{\mathcal{DEP}}\langle\iota_b, \sigma, \overline{\sigma}\rangle$ corresponds with an (internal) signal that represents the creation of a new deployment (the ι_b represents a unique identifier for every branching point, σ represents the operator signal, and $\overline{\sigma}$ the operand signals).

3.2.4 Configurations. The state of a turn of a reactive program is thus modelled as a configuration k which consists of a value environment E containing the values of the external signals, a set of deployment identifiers that are active in the current turn I_d (i.e. to model toggle semantics), a set of deployment wirings W, a set of deployment snapshots S (of the active deployments) and a toggle environment D (which stores, for every branching point ι_b the captures, as identified by ι_c that it has already instantiated into a deployment ι_d).

3.2.5 Primitive Operations. Finally, we assume a set of primitives that are available to the operational semantics. Every primitive (denoted as a p), has a corresponding function δ_p. We assume that every δ_p is total (w.r.t. the values $\in V$).

3.3 Initial Configuration

Figure 4 shows how the initial configuration is constructed, given a reactive program p. The initial configuration k_{init} contains an initial wiring in W_{init} that deploys the `main` reactor (as defined in p) to bootstrap the reactive program. It also contains a value environment, which contains the bindings for the captures for the reactors defined in p (these are created by applying the function ntoc on each N_i), as well as the initial values of the external signals (such as `time`, ...). The mechanism that determines the later values will be defined later as described in Section 3.5.

In the definition of W_{init} and ntoc an initial signal environment Σ_{init} is used. This environment contains bindings for: 1) The primitive operations. To disambiguate between the symbol + and the object representing its computation, the latter is typeset as $\ulcorner + \urcorner$. We only provide +, but this set can easily be extended with more. 2) The primitive reactors in p (binding their name to an $\mathcal{S}_{\mathcal{GLB}}$. And 3) the external time-varying signals. Similar to the primitive operations, we only assume the existence of *time*. However, this can also easily be extended.

3.4 Reduction Rules

After having presented the syntax and the semantic entities, we present the small-step operational intra-turn semantics. We make a distinction between two types of rules: wiring rules (or w-rules for short) that reconfigure the dependency graph and snapshot rules (or s-rules for short) that determine the values of the signals in a deployment during a given turn. Semantically, both kinds of rules can be executed non-deterministically in different regions of the reactive programs graph. Thus, there is only one reduction relation describing the intra-turn semantics ($\rightarrow_k$). However, as most w-rules only operate on $\mathcal{W}$ objects, and a large part of the s-rules only on $\mathcal{S}$ objects: we define two helper reduction relations ($\rightarrow_w$ and $\rightarrow_s$) that operate only on a single $\mathcal{W}$ or $\mathcal{S}$ object.

3.4.1 Wiring Rules. The reduction rules that perform wiring-level operations are presented in Figure 5. In short, the goal of these rules is to go from a set of nodes (N) to a signal environment (Σ) containing all the signals available to a deployment of a capture.

The W-REF rule replaces one of the named inputs of a node (left-hand side) with the actual signal as stored in Σ. This rule ensures that all the other rules can operate with only signals (including constant signals) as inputs..

The purpose of the W-DEPLOY rule is to create a hole that will be filled in by the s-rules with the actual deployment. In summary, W-DEPLOY makes the following changes to Σ. First, it binds $\mathcal{S}_{\mathcal{DEP}}\langle\iota_b, \sigma, \overline{\sigma}\rangle$ to a unique name x. This x will be reduced, by the s-rules, into a ι_d. The ι_b uniquely identifies the hole. And secondly, for every output o_i in $\overline{o}$, o_i will be bound to $\mathcal{S}_{\mathcal{REF}}\langle\mathcal{S}_{\mathcal{REF}}\langle\iota_d, x\rangle, out_{i,0}\rangle$. The inner $\mathcal{S}_{\mathcal{REF}}$ will later be reduced to the ι_d that the $\mathcal{S}_{\mathcal{DEP}}$ bound to x reduces to, and the outer $\mathcal{S}_{\mathcal{REF}}$ will then be reduced into the correct outgoing (sink) value of that deployment.

The W-RHO rule creates a new capture (with a fresh, unique, ι_c) that contains the current environment (with bindings for the signals that N depends on). To let the nested DAGs have access to the sources and sinks of the surrounding environment, the W-RHO rule renames all input and output signals in Σ by incrementing its second index (which denotes the nesting level).

Finally, the W-CONGRUENCE rule connects the local w-rules ($\rightarrow_w$) to the global reduction relation ($\rightarrow_k$). This is the only global w-rule as the purpose of the w-rules is to configure the dependency graph without using the *current* values of any signals, and that no w-rule needs to interact with another deployment. Thus, there is no need for other global w-rules.

3.4.2 Snapshot Rules. Figure 6 shows the snapshot reduction rules. These rules describe how a signal environment of a deployment is reduced into a value environment. Most of these rules make use of $\mathcal{E}$ which defines an evaluation context for signal environments. I.e. the reduction rules listed in Figure 6 are performing a kind of *graph reduction*. The evaluation context $\mathcal{E}$ therefore looks for a (sub)expression for a signal present in Σ which can then be reduced using the s-rules.

The S-SELF-REF rule replaces a $\mathcal{S}_{\mathcal{REF}}$ refering to a local name with the result of looking up that name in its own environment (only if it is a value, and not a yet-to-be-reduced signal).

The S-DEPLOY-PRIMITIVE and S-TUPLE-REF rules are used for the primitive operations. The functions that model the primitive operations (i.e. the δ_p functions) always return multiple values. Thus the S-DEPLOY-PRIMITIVE rule produces a list of values ($\in V*$) and, the S-TUPLE-REF is therefore used to get a value from the list (depending on the index i of $out_{i,0}$).

Similar to W-CONGRUENCE, the S-CONGRUENCE rule connects the local s-rules to the global reduction relation $\rightarrow_k$.

[5] This is also encoded as a signal to support higher-order deployments: i.e. the deployment (in which a named signal is referenced by) can be time-varying.

Given $p = \{\mathcal{R}\langle x_1, N_1\rangle, \ldots, \mathcal{R}\langle x_{|p|}, N_{|p|}\rangle\}$ ($|p|$ is the total number of user-defined reactor definitions):

$$\begin{array}{lcl}
k_{init} & = & \mathcal{K}\langle E_{init}, \{\iota_{d,main}\}, W_{init}, \varnothing, \varnothing\rangle \\
E_{init} & = & \{x_i \mapsto \text{ntoc}(N_i) \mid \forall i \in [1..|p|]\} \cup \{time \mapsto 0, \ldots\} \\
W_{init} & = & \{\mathcal{W}\langle \iota_{d,main}, \{([main], \mathcal{DEPLOY}, [out_{(i,0)} \mid \forall i \in [1..|o|_{main}]])\}, \Sigma_{init}\rangle\} \\
\Sigma_{init} & = & \{+ \mapsto \ulcorner + \urcorner, \ldots\} \cup \{x_i \mapsto \mathcal{S}_{\mathcal{GLB}}\langle x_i\rangle \mid \forall i \in [1..|p|]\} \cup \{time \mapsto \mathcal{S}_{\mathcal{GLB}}\langle time\rangle, \ldots\}
\end{array}$$

Where $|o|_{main}$ is the number of outputs defined in the `main` reactor in p, and $\text{ntoc}(N) = \mathcal{C}\langle \iota_c, N, \Sigma_{init}\rangle$ (where ι_c fresh)

Figure 4: Initial configuration, given a reactive program *p*.

Local Wiring Rules ($\rightarrow_w$):

(W-REF)

$$\mathcal{W}\langle \iota_d, \{(\bar{i}_l \,\S\, [x] \,\S\, \bar{i}_r, nt, \bar{o})\} \uplus N, \Sigma\rangle \rightarrow_w \mathcal{W}\langle \iota_d, \{(\bar{i}_l \,\S\, [\Sigma(x)] \,\S\, \bar{i}_r, nt, \bar{o})\} \cup N, \Sigma\rangle$$

(W-DEPLOY)

$$\frac{\begin{array}{c} x, \iota_b \text{ fresh} \\ \Sigma' = \Sigma[x \mapsto \mathcal{S}_{\mathcal{DEP}}\langle \iota_b, \sigma, \overline{\sigma}\rangle][o_i \mapsto \mathcal{S}_{\mathcal{REF}}\langle \mathcal{S}_{\mathcal{REF}}\langle \iota_d, x\rangle, out_{i,0}\rangle \mid \forall i \in [1..|\overline{o}|]] \end{array}}{\mathcal{W}\langle \iota_d, \{([\sigma] \,\S\, \overline{\sigma}, \mathcal{DEPLOY}, \overline{o})\} \uplus N, \Sigma\rangle \rightarrow_w \mathcal{W}\langle \iota_d, N, \Sigma'\rangle}$$

(W-RHO)

$$\frac{\iota_c \text{ fresh} \qquad \Sigma_c = \text{shift\_io}(\Sigma) \qquad c = \mathcal{C}\langle \iota_c, N_{inner}, \Sigma_c\rangle}{\mathcal{W}\langle \iota_d, \{(\overline{\sigma}, \mathcal{RHO}\langle N_{inner}\rangle, [o])\} \uplus N, \Sigma\rangle \rightarrow_w \mathcal{W}\langle \iota_d, N, \Sigma[o \mapsto c]\rangle}$$

Where $\text{shift\_io}(\Sigma) = \{x' \mapsto \sigma \mid \forall x \mapsto \sigma \in \Sigma, x' = \begin{cases} in_{i,j+1} & x = in_{i,j} \\ out_{i,j+1} & x = out_{i,j} \\ x & \text{otherwise} \end{cases}\}$

Global Rules ($\rightarrow_k$):

(W-CONGRUENCE)

$$\frac{w \rightarrow_w w'}{\mathcal{K}\langle E, I_d, \{w\} \uplus W, S, D\rangle \rightarrow_k \mathcal{K}\langle E, I_d, \{w'\} \cup W, S, D\rangle}$$

Figure 5: Wiring (graph construction) rules of Karcharias. The § symbol denotes concatenation.

Evaluation Contexts:

$$\begin{array}{lcl}
\mathcal{E} & ::= & \{x \mapsto \mathcal{E}_\sigma\} \uplus \Sigma \\
\mathcal{E}_\sigma & ::= & \square \\
 & \mid & \mathcal{S}_{\mathcal{REF}}\langle \mathcal{E}_\sigma, x\rangle \\
 & \mid & \mathcal{S}_{\mathcal{DEP}}\langle \iota_b, \mathcal{E}_\sigma, \overline{\sigma}\rangle \\
 & \mid & \mathcal{S}_{\mathcal{DEP}}\langle \iota_b, p, \overline{\sigma} \,\S\, [\mathcal{E}_\sigma] \,\S\, \overline{\sigma}\rangle
\end{array}$$

Local Snapshot Rules ($\rightarrow_s$):

(S-SELF-REF)

$$\frac{\Sigma = \mathcal{E}[\mathcal{S}_{\mathcal{REF}}\langle \iota_d, x\rangle] \qquad v = \Sigma(x)}{\mathcal{S}\langle \iota_d, \Sigma\rangle \rightarrow_s \mathcal{S}\langle \iota_d, \mathcal{E}[v]\rangle}$$

(S-DEPLOY-PRIMITIVE)

$$\mathcal{S}\langle \iota_d, \mathcal{E}[\mathcal{S}_{\mathcal{DEP}}\langle \iota_b, p, \overline{v}\rangle]\rangle \rightarrow_s \mathcal{S}\langle \iota_d, \mathcal{E}[\delta_p(\overline{v})]\rangle$$

(S-TUPLE-REF)

$$\mathcal{S}\langle \iota_d, \mathcal{E}[\mathcal{S}_{\mathcal{REF}}\langle \overline{v}, out_{(0,i)}\rangle]\rangle \rightarrow_s \mathcal{S}\langle \iota_d, \mathcal{E}[v_i]\rangle$$

Global Rules ($\rightarrow_k$):

(S-CONGRUENCE)

$$\frac{s \rightarrow_s s'}{\begin{array}{c}\mathcal{K}\langle E, I_d, W, \{s\} \uplus S, D\rangle \\ \rightarrow_k \mathcal{K}\langle E, I_d, W, \{s'\} \cup S, D\rangle\end{array}}$$

(S-ACTIVATE)

$$\frac{\begin{array}{c}\mathcal{W}\langle \iota_d, \varnothing, \Sigma\rangle \in W \qquad \iota_d \in I_d \\ \forall s \in S : s = \mathcal{S}\langle \iota'_d, \Sigma'\rangle, \iota_d \neq \iota'_d\end{array}}{\begin{array}{c}\mathcal{K}\langle E, I_d, W, S, D\rangle \\ \rightarrow_k \mathcal{K}\langle E, I_d, W, \{\mathcal{S}\langle \iota_d, \Sigma\rangle\} \cup S, D\rangle\end{array}}$$

(S-REF)

$$\frac{\Sigma = \mathcal{E}[\mathcal{S}_{\mathcal{REF}}\langle \iota'_d, x\rangle] \qquad v = \Sigma'(x) \qquad \Sigma'' = \mathcal{E}[v]}{\begin{array}{c}\mathcal{K}\langle E, I_d, W, \{\mathcal{S}\langle \iota_d, \Sigma\rangle, \mathcal{S}\langle \iota'_d, \Sigma'\rangle\} \uplus S, D\rangle \\ \rightarrow_k \mathcal{K}\langle E, I_d, W, \{\mathcal{S}\langle \iota_d, \Sigma''\rangle, \mathcal{S}\langle \iota'_d, \Sigma'\rangle\} \cup S, D\rangle\end{array}}$$

(S-GLOBAL)

$$\frac{\Sigma = \mathcal{E}[\mathcal{S}_{\mathcal{GLB}}\langle x\rangle] \qquad \Sigma' = \mathcal{E}[E(x)]}{\begin{array}{c}\mathcal{K}\langle E, I_d, W, \{\mathcal{S}\langle \iota_d, \Sigma\rangle\} \uplus S, D\rangle \\ \rightarrow_k \mathcal{K}\langle E, I_d, W, \{\mathcal{S}\langle \iota_d, \Sigma'\rangle\} \cup S, D\rangle\end{array}}$$

(S-DEPLOY-NEW)

$$\frac{\begin{array}{c}\Sigma = \mathcal{E}[\mathcal{S}_{\mathcal{DEP}}\langle \iota_b, \mathcal{C}\langle \iota_c, N, \Sigma_c\rangle, \overline{\sigma}\rangle] \qquad (\iota_b, \iota_c) \notin \text{dom}(D) \qquad \iota'_d \text{ fresh} \\ w = \mathcal{W}\langle \iota'_d, N, \Sigma_c[in_{i,0} \mapsto \sigma_i \mid \forall i \in [1..|\overline{\sigma}|]]\rangle \qquad \Sigma' = \mathcal{E}[\iota'_d] \\ D' = D[(\iota_b, \iota_c) \mapsto \iota'_d]\end{array}}{\begin{array}{c}\mathcal{K}\langle E, I_d, W, \{\mathcal{S}\langle \iota_d, \Sigma\rangle\} \uplus S, D\rangle \\ \rightarrow_k \mathcal{K}\langle E, \{\iota'_d\} \cup I_d, \{w\} \cup W, \{\mathcal{S}\langle \iota_d, \Sigma'\rangle\} \cup S, D'\rangle\end{array}}$$

(S-DEPLOY-EXISTING)

$$\frac{\Sigma = \mathcal{E}[\mathcal{S}_{\mathcal{DEP}}\langle \iota_b, \mathcal{C}\langle \iota_c, N, \Sigma_c\rangle, \overline{\sigma}\rangle] \qquad \iota'_d = D(\iota_b, \iota_c) \qquad \Sigma' = \mathcal{E}[\iota'_d]}{\begin{array}{c}\mathcal{K}\langle E, I_d, W, \{\mathcal{S}\langle \iota_d, \Sigma\rangle\} \uplus S, D\rangle \\ \rightarrow_k \mathcal{K}\langle E, \{\iota'_d\} \cup I_d, W, \{\mathcal{S}\langle \iota_d, \Sigma'\rangle\} \cup S, D\rangle\end{array}}$$

Figure 6: Snapshot reduction (propagation) rules of Karcharias.

In the presence of toggle semantics, deployments can only be activated (i.e. creating an $\mathcal{S}$ from a $\mathcal{W}$) if and only if the corresponding $\mathcal{W}$ is complete (i.e. its signal environment Σ has been fully populated, which is the case once N is empty) and the deployment is supposed to be active in the current turn. The S-ACTIVATE rule does exactly that: when $\iota_d \in I_d$ (the set of active deployments), and no existing $\mathcal{S}$ exists for ι_d (in the current turn), a new $\mathcal{S}$ is created and added to S if the corresponding $\mathcal{W}$ is complete. Note that it is impossible for a ι_d to be in I_d without there being a corresponding $\mathcal{W}$ (see the description of the S-DEPLOY-NEW and S-DEPLOY-EXISTING rules).

The S-REF rule (the global counterpart of S-SELF-REF) replaces a $\mathcal{S}_{\mathcal{REF}}$ with a value from a different deployment.

The S-GLOBAL rule replaces a $\mathcal{S}_{\mathcal{GLB}}$ with its value as stored in E.

The S-DEPLOY-NEW is the most important rule of our formalisation. Its purpose is to deploy: i.e. create a new $\mathcal{W}$ for a $\mathcal{S}_{\mathcal{DEP}}$ that is present in a signal. It does so by first checking if the capture has already been deployed earlier for the same branching point (i.e. $(\iota_b, \iota_c) \in \text{dom}(D)$). If so, the S-DEPLOY-EXISTING rule will be used instead (which is explained next). If not, it is going to extend the lexical environment stored in the capture with bindings for the source signals as present in $\mathcal{S}_{\mathcal{DEP}}$ and then use it to create a $\mathcal{W}$ for that deployment. D is also updated to remember the deployment.

The final rule is the S-DEPLOY-EXISTING which is used if there exists already a deployment given the branching point and capture stored in the operator position. Instead of allocating a new deployment, it looks up the old deployment ι_d. The old wiring and any accumulated state will thus be reused.

In these last two rules, we show that deployments are instantiated with the argument signals ($\overline{\sigma}$) and not the *current* values of these signals. This mechanism ensures that in a next turn, no additional work is necessary to ensure that a deployment is reacting to the correct values, as in the new turn the lookup is performed again if the deployment is active. Note that this is not the case for

Semantic Entities:

$$\begin{array}{rcl} k^{inter} \in \textbf{Inter-Turn Configuration} & ::= & \mathcal{K}^{inter}\langle \tau, k\rangle \\ \tau \subseteq \textbf{Primitive Time-Varying Sources} & ::= & \{x \mapsto \overline{v}, \ldots\} \\ \widetilde{k} \in \widetilde{K} \subset K & \text{where} & \widetilde{k} \not\to_k \end{array}$$

Initial Configuration:

$$k^{inter}_{init} = \mathcal{K}^{inter}\langle \tau_{init}, k_{init}\rangle$$

$$\tau_{init} = \{time \mapsto [0, 1, 2, 3, \ldots],\ \ldots\}$$

Reduction Relation ($\rightsquigarrow$):

$$\text{(INTRA-TURN)}\quad \frac{k \to_k k'}{\mathcal{K}^{inter}\langle \tau, k\rangle \rightsquigarrow \mathcal{K}^{inter}\langle \tau, k'\rangle}$$

$$\text{(NEXT-TURN)}\quad \frac{\begin{array}{c} k = \mathcal{K}\langle E, I_d, W, S, D\rangle \in \widetilde{K} \\ k' = \mathcal{K}\langle E[x_i \mapsto v_{i,now} \mid \forall i \in [1..|\tau|]], \{\iota_{main,d}\}, W, \varnothing, D\rangle \end{array}}{\begin{array}{c} \mathcal{K}^{inter}\langle \{x_i \mapsto [v_{i,now}] \S\, \overline{v}_i \mid \forall i \in [1..|\tau|]\}, k\rangle \\ \rightsquigarrow \mathcal{K}^{inter}\langle \{x_i \mapsto \overline{v}_i \mid \forall i \in [1..|\tau|]\}, k'\rangle \end{array}}$$

Figure 7: The Inter-Turn Semantics of Karcharias.

primitive operators (using S-DEPLOY-PRIMITIVE and S-TUPLE-REF) as primitive deployments do not create new $\mathcal{W}$s.

3.5 Inter-Turn Semantics

We now focus our attention to the inter-turn semantics. In short, the inter-turn semantics describe how the values of the external source signals are supplied and consequently used by the intra-turn semantics to reduce a configuration to one where all deployments $\in I_d$ have a corresponding $\mathcal{S}$ that whose signal environments contain only values.

The inter-turn semantics are presented in Figure 7 as an extension to our current formalisation. We first define an inter-turn configuration which contains all the *future* values of the time-varying signals (τ), and an intra-turn configuration[6]. Furthermore, we define $\widetilde{K}$ as the set of complete configurations, these are configurations in which all deployments snapshots in I_d have been activated and whose signal environment contains only values. Assuming that a configuration does not get stuck prematurily, this is the case when a k is irreducible by $\to_k$. Run-time errors (such as update errors, type errors, arity errors...) are currently not detected in our formalisation.

Just as with the intra-turn semantics, we define an initial inter-turn configuration k^{inter}_{init} which contains k_{init} from Figure 4 and a mapping for the time-varying signals τ_{init}. In practice, the values contained in τ_{init} cannot usually be determined a priori. However, for the sake of simplicity we assume that all values of these signals can be determined beforehand.

The small-step inter-turn semantics are formalised by $\rightsquigarrow$. Just as the intra-turn semantics, the inter-turn semantics are modelled as small-step operational semantics. The INTRA-TURN rule makes the semantics of a turn ($\to_k$) available to the inter-turn semantics. It reduces an incomplete configuration k to a configuration k'. The NEXT-TURN rule inserts the new values of the external time-varying values from τ into the current turn’s final configuration k, in order to create a new k' in which all deployment snapshots have been removed, and where I_d is reset to contain only the bootstrap deployment (as created in Section 3.3).

In short, starting from k^{inter}_{init}, the reactive program is evaluated by reducing INTRA-TURN as much steps as necessary to reach a complete confugration $\widetilde{k}$, followed by one reduction using NEXT-TURN, and then repeating this (possibly) ad infinitum. Remark that the outputs of the `main` reactor are not used by $\rightsquigarrow$. While an actual implementation would need to use these as output (e.g., to actuate), this has not been formalised as we do not consider it part of Karcharias’ core computational model.

[6]Remember from Section 3.3 that the initial values of the external time-varying sources (such as `time`) is already encoded in E, thus only the *next* values of these signals has to be encoded in τ.

4 DISCUSSION

Our formalisation of graph-based RP provides a clear semantic for deployments: i.e. instantiated parts of (dependency) graphs. Deployment nodes in the dependency graph, which can be filled in at run-time, make it possible for a developer to compose reactive programs out of smaller programs, similar to the composition of signal functions in function-based RP. An interesting future research avenue is to further compare the semantics of graph-based RP with the semantics (and implementations) of function-based RP: e.g., to establish whether or not they are computationally equivalent (i.e. can they implement the same RP programs).

Our work presented here does not formalise all possible aspects of an RPL. One noteworthy omission are conditional deployments created using `if`. We argue that semantics are already captured by the higher-order deployments. By “delaying” a graph using $\mathcal{RHO}\langle N\rangle$ and providing an eager δ_{if}, one can easily model conditionals (and thus, conditional deployments) using our current formalisation. In the future, we may prove this formally.

Another noteworthy omission is the support for stateful computations (e.g., using operators like `foldp` [7] or using other state management mechanisms [26]). We argue that doing so is rather trivial. In short, such functionality can be formalised by providing a heap-like store to store turn-transient information in.

Finally, graph-based RPLs usually update their signals incrementally: only signals affected by an (external) change are usually recomputed. This is completely absent from our formalisation as at the start of each turn the NEXT-TURN rule throws away all $\mathcal{S}$s. However, incrementality is, in essence, only an optimisation. It thus should have no impact on the semantics. Hence we argue that there was also no need to explicitely formalise this.

5 CONCLUSION

This paper presented Karcharias, a small-step operational semantics of a graph-based reactive programming language. While our formalisation, as presented in this paper, mainly focuses on the reactive programming paradigm, we have observed many similarities with other streaming-based solutions (e.g., Reactive Extensions [22] and Akka Streams [33]). We hypothesise that our formalisation can be generalised such that it also captures other graph-based reactive-*like* languages.

ACKNOWLEDGMENTS

Bjarno Oeyen is funded by the Research Foundation - Flanders (FWO) under grant number 1S93822N.

REFERENCES

[1] Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. 2021. Diamonds are not forever: liveness in reactive programming with guarded recursion. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434283

[2] Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. 2013. A survey on reactive programming. *ACM Comput. Surv.* 45, 4 (2013), 52:1–52:34. https://doi.org/10.1145/2501654.2501666

[3] Andoni Lombide Carreton, Stijn Mostinckx, Tom Van Cutsem, and Wolfgang De Meuter. 2010. Loosely-Coupled Distributed Reactive Programming in Mobile Ad Hoc Networks. In *Objects, Models, Components, Patterns, 48th International Conference, TOOLS 2010, Málaga, Spain, June 28 - July 2, 2010. Proceedings (Lecture Notes in Computer Science)*, Jan Vitek (Ed.), Vol. 6141. Springer, 41–60. https://doi.org/10.1007/978-3-642-13953-6_3

[4] 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

[5] 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. https://doi.org/10.1007/11693024_20

[6] 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)*, I. V. Ramakrishnan (Ed.), Vol. 1990. Springer, 29–44. https://doi.org/10.1007/3-540-45241-9_3

[7] 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

[8] N.G de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)* 75, 5 (1972), 381–392. https://doi.org/10.1016/1385-7258(72)90034-0

[9] 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, November 15-17, 2020, Berlin, Germany (Virtual Conference) (LIPIcs)*, Robert Hirschfeld and Tobias Pape (Eds.), Vol. 166. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 19:1–19:29. https://doi.org/10.4230/LIPIcs.ECOOP.2020.19

[10] Joscha Drechsler, Ragnar Mogk, Guido Salvaneschi, and Mira Mezini. 2018. Thread-safe reactive programming. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 107:1–107:30. https://doi.org/10.1145/3276477

[11] 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

[12] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex.* MIT Press, London, England.

[13] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, Robert Cartwright (Ed.). ACM, 237–247. https://doi.org/10.1145/155090.155113

[14] John Hughes. 2000. Generalising monads to arrows. *Sci. Comput. Program.* 37, 1-3 (2000), 67–111. https://doi.org/10.1016/S0167-6423(99)00023-4

[15] 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. https://doi.org/10.1007/11737414_18

[16] Wolfgang Jeltsch. 2014. Categorical Semantics for Functional Reactive Programming with Temporal Recursion and Corecursion. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014 (EPTCS)*, Paul Blain Levy and Neel Krishnaswami (Eds.), Vol. 153. 127–142. https://doi.org/10.4204/EPTCS.153.9

[17] Christopher T King. 2008. *An axiomatic semantics for functional reactive programming.* Ph.D. Dissertation. Worcester Polytechnic Institute.

[18] Neelakantan R. Krishnaswami and Nick Benton. 2011. Ultrametric Semantics of Reactive Programs. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada.* IEEE Computer Society, 257–266. https://doi.org/10.1109/LICS.2011.38

[19] Adriaan Leijnse. 2020. An introduction to Denotative Continuous Spacetime Programming. https://2020.splashcon.org/details/rebls-2020-papers/8/An-Introduction-to-Denotative-Continuous-Spacetime-Programming-Work-in-Progress-

[20] Hai Liu and Paul Hudak. 2007. Plugging a Space Leak with an Arrow. *Electron. Notes Theor. Comput. Sci.* 193 (2007), 29–45. https://doi.org/10.1016/j.entcs.2007.10.006

[21] Ingo Maier and Martin Odersky. 2012. Deprecating the Observer Pattern with Scala.React. (2012), 20. http://infoscience.epfl.ch/record/176887

[22] Erik Meijer. 2010. Reactive Extensions (Rx): Curing Your Asynchronous Programming Blues. In *ACM SIGPLAN Commercial Users of Functional Programming* (Baltimore, Maryland) *(CUFP '10).* Association for Computing Machinery, New York, NY, USA, Article 11, 1 pages. https://doi.org/10.1145/1900160.1900173

[23] 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

[24] Henrik Nilsson, Antony Courtney, and John Peterson. 2002. Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell* (Pittsburgh, Pennsylvania) *(Haskell '02).* Association for Computing Machinery, New York, NY, USA, 51–64. https://doi.org/10.1145/581690.581695

[25] Bjarno Oeyen, Sam Van den Vonder, and Wolfgang De Meuter. 2020. Reactive sorting networks. In *REBLS 2020: Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, Virtual Event, USA, November 16, 2020.* ACM, 38–50. https://doi.org/10.1145/3427763.3428316

[26] Bjarno Oeyen, Sam Van den Vonder, and Wolfgang De Meuter. 2021. Trampoline variables: a general method for state accumulation in reactive programming. In *REBLS 2021: Proceedings of the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, Chicago, IL, USA, 18 October 2021*, Louis Mandel (Ed.). ACM, 27–40. https://doi.org/10.1145/3486605.3486787

[27] 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

[28] 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

[29] 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

[30] Neil Sculthorpe and Henrik Nilsson. 2009. Safe functional reactive programming through dependent types. In *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, Graham Hutton and Andrew P. Tolmach (Eds.). ACM, 23–34. https://doi.org/10.1145/1596550.1596558

[31] 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

[32] 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

[33] Typesafe, inc. [n.d.]. Akka: Build concurrent, distributed, and resilient message-driven applications for Java and scala. https://akka.io/

[34] Zhanyong Wan and Paul Hudak. 2000. Functional reactive programming from first principles. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, Monica S. Lam (Ed.). ACM, 242–252. https://doi.org/10.1145/349299.349331

[35] 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)*, Shriram Krishnamurthi and C. R. Ramakrishnan (Eds.), Vol. 2257.

Springer, 155–172. https://doi.org/10.1007/3-540-45587-6_11

[36] Tian Zhao, Adam Berger, and Yonglun Li. 2020. Asynchronous monad for reactive IoT programming. In *REBLS 2020: Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, Virtual Event, USA, November 16, 2020*. ACM, 25–37. https://doi.org/10.1145/3427763.3428314