A Virtual Machine for Higher-Order Reactors

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

Jens Nicolay
Vrije Universiteit Brussel
Brussels, Belgium
jens.nicolay@vub.be

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

ABSTRACT
We present the design of a Remus, a virtual machine for reactive (RP) programs — signal-based programs that react automatically to changes — that is currently under development. The design of Remus has been tailored specifically for reactive programs: it has a dedicated instruction set and a memory model that is compatible with the model of reactive programming. The virtual machine allows reactive programs to be evaluated without needing to compile them to native code, nor by running them on an existing (general-purpose) virtual machine that is not specific to RP. The model of the virtual machine makes it possible to reason over the various behavioural aspects of a reactive program, such as memory consumption, in a language-agnostic way. As an example, we propose a static analysis that aims to eliminate dynamic allocations that occur in dynamic reactive programs: i.e., programs where the dependencies between the signals that make up a program changes at run-time. We show that our virtual machine, supplemented by the results of the analysis, allows for (well-behaved) reactive programs to be evaluated in constant space (i.e. in bounded memory) even when dependencies change or new signals are created programmatically.

CCS CONCEPTS
• Software and its engineering → Data flow languages; Virtual machines; • Computer systems organization → Real-time languages.

KEYWORDS
Reactive Programming, Virtual Machine, Static Compilation, Higher-Order Programming

ACM Reference Format:

1 INTRODUCTION
The reactive programming (RP) model proposes a declarative implementation style for implementing event-driven applications [1]. Whenever a new data event (e.g., a reading from a physical sensor, an interaction in a user interface, or a message received over a network) is received by an RP program, the reactive runtime automatically reacts to this change. Programs are written by defining so-called signals that are expressed declaratively. Whenever a new data event arrives into the system, the local state of every signal (i.e. its current value) is updated by the language runtime.

Many of the RP language implementations that exist today are conceived as an Embedded Domain Specific Language (EDSL): e.g., Yampa [4] is built on top of Haskell, REScala [7] on top of Scala, FrTime [2] on top of Racket...Lifting, an essential mechanism found in any EDSL RP language, allows host language code to become integrated directly in the reactive program. As such, the reactive program has access to the full feature set provided by the host language. Furthermore, EDSL RP languages allow existing tools designed for the host language — like compilers, debuggers, interpreters — to be re-used. This means that an RP language can get any functionality and optimisations intended for the host language, often with minimal effort.

Nonetheless, the embedded approach also has a number of disadvantages due to the paradigmatic mismatch that occurs when reactive and non-reactive code are intertwined [13]. In this paper, we will focus on the mismatch in how RP programs allocate memory with respect to general-purpose languages. In short, we claim that the memory model for a general purpose host language is not a great fit for RP. EDSL RP languages often re-use the memory manager of the host language as-is. This design decision — by the RP language designers — is obvious, as the RP program needs to call host language code due to lifting, and thus its memory manager is considered essential. However, in many cases (i.e. when no dependencies between signals change at-runtime), an RP program requires only constant space to keep track of signal updates. Thus, the RP model itself has no need for a complex memory manager. In brief, embedding reactivity into a non-reactive model results in complex interactions between the reactive and non-reactive paradigms.

For these reasons, we have designed and implemented a pure RP language [5], a language that is not embedded in another language, where all code is reactive. Nevertheless, our prototypical implementation is an expression-based interpreter where signals are directly created and manipulated, similar to many EDSL RP language implementations. Therefore, programs evaluated by our interpreter still rely on Racket’s run-time facilities — such as its memory management with garbage collection — even if programs written in our language do not have any access to Racket code. Given this current state of affairs, we have thus decided to design a dedicated runtime environment for our language. I.e., we have designed a virtual machine — dubbed Remus — with a dedicated instruction set and a memory model that is tailored for the execution of reactive programs.
we present the general design of the virtual machine: its memory
As such, the (e.g., if the user has pressed a button) this behaviour may change to
the value of its second operand.
duces a non-false value, the output of the
formats temperature readings — is shown in listing
morphic over time
A reactor means providing some actual source signals as arguments, which in turn creates new signals, some of which are the reactor’s sinks. As Haai is pure, even the + primitive is modelled as a reactor, i.e. as a reactor that creates a signal producing the addition of the values produced by other signals. Listing 1 provides an example of a Haai reactor that computes the average of two numbers. Haai processes changes to signals in logical turns, i.e. without glitches [2]. Every reaction to a received data event is executed to completion, in a way that each signal is updated at most once.
Reactors in Haai are first-class: signals can emit a reactor as a value. When such a signal is used in operator position of a deployment expression, this deployment expression becomes polymorphic over time. An example of a higher-order reactor — that formats temperature readings — is shown in listing 2. By default, it alternates between showing the temperature in either °C or °F, toggling every second between the two. However, in some situations (e.g., if the user has pressed a button) this behaviour may change to a specific format which is determined by the r-override source. As such, the r signal in listing 2 will emit the actual reactor to use for formatting.

The contributions discussed in this paper are as follows. First, we present the general design of the virtual machine: its memory model and its set of instructions used to denote the flow of data in RP programs. Second, we propose a simple static analysis in terms of our virtual machine in which a derivative instruction set can be derived such that all memory needed by the RP program can be allocated preemptively. In other words, once the RP program has started, it no longer needs to allocate any additional memory from the underlying system. This contribution is important to make RP safe and viable option for building long-running reactive applications in constrained memory environments.

2 A REACTIVE VIRTUAL MACHINE

2.1 Underlying Evaluation Model

The design of Remus is based on Haai [5], a paradigmatically pure RP language. Haai uses reactors as the main unit of composition: instead of lifting host language functions to create signals, Haai programs are composed out of pure reactors. Reactors are, in essence, graphs with sources and sinks whose instantiation (deployment) creates the actual time-varying signals. I.e., instantiating (deploying) a reactor means providing some actual source signals as arguments, which in turn creates new signals, some of which are the reactor’s sinks. As Haai is pure, even the + primitive is modelled as a reactor, i.e. as a reactor that creates a signal producing the addition of the values produced by other signals. Listing 1 provides an example of a Haai reactor that computes the average of two numbers. Haai processes changes to signals in logical turns, i.e. without glitches [2]. Every reaction to a received data event is executed to completion, in a way that each signal is updated at most once.

Reactors in Haai are first-class: signals can emit a reactor as a value. When such a signal is used in operator position of a deployment expression, this deployment expression becomes polymorphic over time. An example of a higher-order reactor — that formats temperature readings — is shown in listing 2. By default, it alternates between showing the temperature in either °C or °F, toggling every second between the two. However, in some situations (e.g., if the user has pressed a button) this behaviour may change to a specific format which is determined by the r-override source. As such, the r signal in listing 2 will emit the actual reactor to use for formatting.

Listing 2: Reactor that formats a temperature according to time, or according to a specified procedure.

Note that the or in Haai works similar to the or in Racket: if the first operand produces a non-false value, the output of the or will be that value, else the or produces the value of its second operand.

2.2 Virtual Machine Design

We now present the virtual machine for our reactor-based language. A program in the VM corresponds to a set of compiled reactor definitions. Note here that one of these reactors will be used as an entry point by the VM (the main reactor). This is the reactor whose deployment corresponds to the actual program.

Reactors from Haai are compiled into two separate instruction sequences. The first describes the steps needed to initialise a reactor deployment. We call these the deployment-time instructions. For the average-of-two reactor from (listing 1) this sequence will contain instructions that deploy + and / . The second instruction sequence describes the at-reaction time behaviour of a reactor: it describes the steps needed to propagate values through the signals of a deployment of the reactor. We call these the reaction-time instructions. E.g., for average-of-two this sequence will contain instructions that move data to (and from) the + and / deployments. Examples of these instruction sequences will provided later in section 2.4, we describe the instruction set and memory model first.

2.3 Instruction Set and Memory Model

The virtual machine is neither stack-based nor register-based: the virtual machine’s job is to assign a value to each signal — in every turn — and as such deployments of reactors will have dedicated slots to store any intermediate values of any signals. To keep the design of the VM simple, most instructions will correspond with a signal definition in the original RP program, and each deployment will provide a memory address for every instruction to store the value of the corresponding signal. Note that one reactor can be deployed more than once, as such each deployment will provide memory for every signal that it contains. If not, any stateful computations contained in such a deployment becomes faulty.

Figure 1 presents the instruction set (the mnemomics). There are different types of operands in use that determine whether something is a literal (constant) value (denoted by o), a local address referring to a value in the same deployment (denoted by l), a reactor or signal identified by name (denoted by n), or a literal number (denoted by n). Some instructions have multiple variants that can have either a literal values or a location as an operand, which are then denoted by o.

We now discuss each instruction. Note that when we describe that an instruction “stores a value” — without any further qualification as to where — we mean that it will be stored in the location associated with that instruction.

• The DeployMono instruction deploys a single reactor by name. It stores the address of this deployment.
• The MakePoly and DeployPoly instructions are used for polymorphic deployments. MakePoly creates a blank reactor table. A reactor table associates reactors with their deployments. MakePoly stores the address of this table. DeployPoly then uses this table to determine the address of the reactor to use at each reaction. The first operand must therefore refer to the corresponding MakePoly instruction and the reactor’s name should be found from the address in the second operand. If it does not exist — which is the case the first time — a new deployment will be created at run-time and its address will be stored in the table for future turns. DeployPoly stores the address of the created (or re-used) deployment. The DeployPoly instruction stores the address of the new (or reused) deployment.

• The Global instruction stores the value of a global signal. The VM provides, via this instruction, access to any supported external data event streams.

• The Supply, React, and Consume instructions move values from one deployment to another, and let a nested deployment react by invoking its instructions. The Supply instruction supplies a local value: the first operand determines which value (which can be either a literal or a local value identified by an address), the second operand determines to which deployment (which should point to a local DeployMono or DeployPoly instruction), the third operand determines the index of the source. The React instruction directs the VM to run the reaction instructions of the referred deployment. Its sole operand is the local address in which the address of that deployment is stored. The Consume instruction then takes the value stored in a sink of that deployment. The second operand determines the index of that sink. Only the Consume instruction stores a value (which is the value consumed from the sinks of the deployment referred).

• The Sink instruction stores a local value into the sinks of the reactor. As there are no general purpose “move” instructions, this is the only method how a deployment can update its own sinks.

Note that there are no jump nor branching instructions. The only “jump” performed by the VM happens when the reaction of the main deployment. The new values of any of the external data sources are then sampled, and the program effectively restarts with a new turn, retaining any previously-computed values. Thus — without considering recursive reactors — every turn is naturally guaranteed to terminate, a very favourable property of a reactive program. Without this guarantee, an RP program may diverge if the system’s memory during any turn. To tackle this problem, we propose a static analysis that pre-allocates the polymorphic deployments. In essence, by pre-determining the reactor tables associated with the MakePoly and DeployPoly instructions, combined with

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>R01</td>
<td>Supply(Z, R01, 1)</td>
</tr>
<tr>
<td>R02</td>
<td>React(Z)</td>
</tr>
<tr>
<td>R03</td>
<td>Consume(Z, 1)</td>
</tr>
<tr>
<td>R04</td>
<td>Supply(R04, R02, 1)</td>
</tr>
<tr>
<td>R05</td>
<td>React(R02)</td>
</tr>
<tr>
<td>R06</td>
<td>Consume(R02, 1)</td>
</tr>
<tr>
<td>R07</td>
<td>Sink(R08, 1)</td>
</tr>
<tr>
<td>R08</td>
<td>Sink(R08, 1)</td>
</tr>
<tr>
<td>R09</td>
<td>Sink(R08, 1)</td>
</tr>
</tbody>
</table>

(a) average-of-two

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>R01</td>
<td>DeployMono(a)</td>
</tr>
<tr>
<td>R02</td>
<td>DeployMono(b)</td>
</tr>
<tr>
<td>R03</td>
<td>DeployPoly(o)</td>
</tr>
<tr>
<td>R04</td>
<td>MakePoly</td>
</tr>
</tbody>
</table>

(values 0 and 1 are pre-allocated)

(b) format-temp

Figure 2: Compiled reactor byte-code

(DEPLOYMONO), two that supply the values of a and b (2 × SUPPLY), one that lets the nested deployment react (REACT), and one to consume its sink value (CONSUME).

This approach of compiling individual reactors is the main advantage over existing RP implementations that target embedded devices [9, 11] which compile a reactive program in its entirety. While jumping from one reactor’s instruction to another does introduce a computational overhead, this approach allows for easily switching to a different reactor (over time) by simply using another reactor’s instructions.

First-Order Compilation Example. The result of compiling listing 1 is shown in fig. 2a. As convention, we use the D and R as a prefix for the addresses of deployment-time instructions and reaction-time instructions, and I and O for sources (inputs) and sinks (outputs). There are only two deployment-time instructions, which supply the + and / reactors, respectively. The reaction-time instructions then move data from the sources (identified by operands prefixed with I) to the nested operators. First, the values of the source signals are supplied to the deployment of +, next this deployment reacts to the supplied values, and then the sum is consumed locally. Afterwards, this sum and the constant 2 is supplied to the deployment of /, this reactor reacts to the supplied values and the average is then consumed locally.

Higher-Order Compilation Example. Figure 2b shows the compiled version of listing 2. Some instructions were omitted for brevity. It shows how MakePoly and DeployPoly are paired in the byte-code. During the deployment of the format-temp reactor, a blank reactor table is allocated to keep track of the polymorphic deployments. At reaction-time, it will read the value of the reactor stored at offset R13 (i.e. the output of or) and use the reactor contained therein to determine which deployment to use.

3 PRE-DETERMINING REACTOR TABLES

In the version of the VM discussed so far, polymorphic deployments are delayed until they are actually needed at run-time. As a consequence, new reactors can be deployed in any turn — not just the first — and therefore, in an execution environment in which memory is constrained, it is possible for the program to exhaust the system’s memory during any turn. To tackle this problem, we propose a static analysis that pre-allocates the polymorphic deployments. In essence, by pre-determining the reactor tables associated with the MakePoly and DeployPoly instructions, combined with
an adaptation of the VM where the DeployPoly instruction is replaced by one that only needs to toggle, not allocate, deployments. RP programs no longer need to allocate at reaction-time.

3.1 Toggle Set Analysis
The Toggle Set Analysis computes the so-called toggle set of every signal: i.e. the set of reactors that it may emit at run-time and that a polymorphic deployment needs to make deployments for. It is a top-down whole-program analysis [6] that starts with a deployment of the main reactor and exploratively detects all polymorphic deployments (from the MakePoly and DeployPoly instructions). For these signals, one needs to determine all possible reactor values that may flow over these signals by analysing the dataflow of the program. When the values of multiple signals may be combined (e.g., if), the toggle sets of those signals are merged. The analysis is expected to be sound: if a reactor may be emitted by a signal at run-time, it must be included in its computed toggle set. To remain sound, it may emit false positives [3].

Once all toggle sets have been computed, the reactor tables can be statically generated. Furthermore, if a signal’s toggle set is determined to contain just a single reactor, the polymorphic deployment can be optimised into a simple monomorphic deployment (as long as it is the same for all deployments of that reactor). We remark that these reactor tables are generated per deployment: as each deployment may contain signals with different toggle sets.

3.2 Revised Instruction Set
Figure 3 contains the revised instruction set, both the DeployMono and MakePoly instructions have been removed: all deployments and reactor tables are now loaded during the initialisation of the program and no longer need to be created dynamically. The DeployPoly has also been removed, and has been replaced by a TogglePoly instruction. This difference in name highlights the fact that it does not allocate a deployment. All deployments are allocated when the program starts. Reactor tables for every reactor are identified by a number.

3.3 Example
The new compiled version of format-temp is shown in fig. 4. The differences here correspond with the changes to the instruction set: the DeployMono and MakePoly instructions have been removed, and the DeployPoly instruction has been replaced with a TogglePoly instruction. For brevity, we do not show any reactor tables: the only interesting part — for our current discussion — is which reactors have an entry. The reactor table of a deployment of format-temp is dependent on how it is being deployed. If format-temp is deployed with #f (false) as its second argument, the reactor table contains only entries for the two reactors present in its body. If, however, format-temp is deployed on another signal, the reactor table will be extended with entries for the toggle set of that signal.

3.4 Discussion
The pre-generation of reactor tables allows deployments to be allocated preemptively: one can use the result of the analysis to pre-determine if a specific device has enough memory to run an RP program. However, the result of the analysis is not limited to just running RP programs on Remus: indeed, the intermediate program representation of Remus, combined with the output of the analysis, can be used to compile reactive programs to other forms (e.g., to native code or as bytecode for another VM). For example, one may compile a reactive program further to the JVM, ensure that all needed deployments are allocated during the program’s start up phase, and then run it using the Epsilon GC [10] to get a similar memory allocation behaviour. However, we claim that a tailored VM for RP programs is necessary. While we have focused on the allocation of memory in this paper, we foresee that a tailored VM for RP — like Remus — is beneficial in other ways as well. E.g., we foresee specific extensions to Remus that allow for the creation of a dedicated debugger for RP (similar to [8]). If RP programs would be further compiled to another VM, the realization of such facilities become inherently more challenging.

4 CONCLUSIONS AND FUTURE WORK
We have presented the design of Remus: a virtual machine with a dedicated instruction set architecture for running RP programs. The virtual machine is basic, yet has support for many features important to RP languages. Furthermore, we discussed a simple static analysis that allowed us to pre-determine reactor tables such that programs can be evaluated — by the VM — without requiring a complex memory manager (e.g., with garbage collection facilities).

In the future, we aim to extend the design of our virtual machine to accommodate different kinds of reactive programming languages (besides Haai), and we aim to validate the performance improvements of various kinds of optimisations (such as the pre-generation of the reactor tables).

ACKNOWLEDGMENTS
Bjarno Oeyen was funded by Innoviris under grant 2021-RDIR-12a.
REFERENCES


