An Efficient Runtime System for Reactive Programming

Eine effiziente Laufzeitumgebung für die reaktive Programmierung
Master-Thesis von Malte Viering
Tag der Einreichung:

1. Gutachten: Prof. Dr.-Ing. Mira Mezini
2. Gutachten: Dr. Guido Salvaneschi

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Software Technology Group

An Efficient Runtime System for Reactive Programming
Eine effiziente Laufzeitumgebung für die reaktive Programmierung

Vorgelegte Master-Thesis von Malte Viering

1. Gutachten: Prof. Dr.-Ing. Mira Mezini
2. Gutachten: Dr. Guido Salvaneschi

Tag der Einreichung:

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 1. August 2015

(M. Viering)

Abstract

Reactive Programming (RP) is a new programming language paradigm which specifically targets reactive applications. Over the last few years, more and more companies and developers adopted RP to develop Web applications, user interfaces and asynchronous event-based software. RP has become popular especially in the JavaScript community where libraries like Angular.js and Bacon.js have embraced the reactive paradigm supporting automated propagation of changes.

It has been shown that RP enables the developer to implement reactive applications in a composable way, abstracting over implementation details such as data dependency detection and making reactive software easier to understand. However, in most cases RP abstractions introduce significant performance costs in comparison to the Observer design pattern. Also, memory requirements are significantly higher.

In this thesis, we apply just-in-time compilation (JIT) to improve the performance of RP. We implemented an RP extension to Ruby based on the Truffle language framework. JIT compilation is provided by the Graal Virtual Machine (VM). Our evaluation shows that RP combined with JIT compilation can achieve a runtime performance comparable to the Observer design pattern.

Zusammenfassung

Reaktive Programmierung (RP) ist ein neues Programmierparadigma für die Entwicklung von reaktiven Anwendungen. In den letzten Jahren haben immer mehr Firmen und Entwickler die reaktive Programmierung für die Entwicklung von Webanwendungen, Benutzeroberflächen und asynchroner ereignisorientierter Software übernommen. Besonders in der JavaScript Community ist die reaktive Programmierung sehr populär geworden. Dort haben Bibliotheken wie Anular.js und Bacon.js dieses Programmierparadigma übernommen.

Für die reaktive Programmierung wurde gezeigt, dass sie es Entwicklern ermöglicht, reaktive Anwendungen in einer zusammensetzbaren Art zu entwickeln, über Entwicklungsdetails wie die Datenabhängigkeitserkennung abstrahiert und reaktive Anwendungen besser verständlich macht. Allerdings erhöhen die Abstraktionen, welche die reaktive Programmierung einführt, in der Regel die Laufzeit signifikant im Vergleich zum Beobachter-Entwurfsmuster.

In dieser Arbeit verwenden wir Just-in-time-Kompilierung (JIT-Kompilierung) um die Performance der reaktiven Programmierung zu verbessern. Wir haben Ruby um die reaktive Programmierung erweitert, hierfür haben wir das Truffle Sprachentwicklungs-Framework verwendet. Die JIT-Kompilierung wird von der Graal Virtual Machine (VM) bereitgestellt. Unsere Evaluation zeigt, dass die reaktive Programmierung mit JIT-Kompilierung eine ähnliche Laufzeitperformance wie das Beobachter-Entwurfsmuster erzielen kann.

Contents

List of Figures

List of Tables

Listings

1 Introduction

Reactive programming is a recent programming paradigm tailored for the development of reactive applications. Reactive programming languages come in different flavors, but the common idea is that developers provide a description of how data flow through an application in a declarative style. The language runtime handles change propagation and recalculation of dependent values when needed.

Despite being a recent approach, reactive programming is already a widely adopted programming model. Among others, Microsoft and Netflix are using reactive programming. From a design standpoint, reactive programming has the goal to reduce the complexity of reactive applications. In addition, reactive programming makes them more maintainable and less error-prone. However, the powerful abstractions provided by this approach introduce a significant runtime cost. Applications written in the reactive style are significantly slower than their counterparts written with the Observer design pattern.

In this thesis, we contribute to address this issue with the development of Reactive Ruby, a new prototypical reactive programming language. Reactive Ruby is an extension of Truffle Ruby, an existing implementation of the Ruby language. Truffle Ruby uses the Truffle framework for the implementation of the Ruby language.

Truffle is a language implementation framework which allows developing interpreters and optimizations for them. Together with the Graal just-in-time (JIT) compiler, Truffle can generate optimized code for the developed languages. As a result of the code generation process achieved by Truffle, the Reactive Ruby runtime can benefit from a number of optimizations. Among others, it offers map fusion and JIT compilation.

The problems with Object-Oriented Programming for Reactive Programming

The traditional way to implement reactive applications is to use the Observer design pattern [25]. However, the Observer design pattern is widely criticized [30, 13, 8]. One of the problems is that it mixes the propagation logic and the application logic. The Observer design patten relies on callbacks for its implementation. Moreover, it uses side effects to change the program state during propagation. These side effects make it hard to understand reactive applications because data and control flow are mixed. Another problem is that the Observer design pattern does not provide functionalities to compose reactions – callbacks return void. Unfortunately, this issue appears frequently in practice.

This becomes obvious in the example of the drag and drop functionality. In this case it is necessary to observe the mouse down event, the mouse up event and the mouse move event. Therefore, for solving this simple problem it is already necessary to compose at least 3 observables.

Since the Observer design pattern has no functionality to compose several observables, the developer needs to compose them manually. However, the implementation of several composed observables is complex. That is because the observers are manipulating the program state which other observer also work with.

If the propagation order is important, the composition becomes especially difficult. That is the case because there is no ordering when the Observer design pattern executes a call back. Therefore, the developer needs to build a state machine to handle the unordered notifications.

All these issues concerning the Observer design pattern make it difficult to implement and maintain a reactive application. This is a well-known issue in industry. As Maier points out [30], Adobe has reported that almost half of the reported bugs of desktop applications are related to the event handling logic [35]. Some authors are going as far as to call problems related to the Observer design pattern the *callback hell* [18].

There have been several attempts to fix these problems. Some domains are using data flow languages, like network programming for example [10]. Event-driven programming (EDP) and aspect-oriented programming (AOP) can also be employed to simplify reactive applications [26]. However, both EDP and AOP still use inversion of control and therefore inherit some problems, which are also present in the Observer design pattern [38].

Functional reactive programming (FRP) and reactive programming provide a solution to theses problems. They enable a declarative way of writing reactive applications. Furthermore, they do not rely on inversion of control to develop reactive applications. Initially, FRP was introduced in Haskell, where it provided a good abstraction to implemented GUIs [19]. Later Flapjax [34] showed that these ideas transfer to the development of web applications in the mainstream language JavaScript.

Reactive programming provides the following characteristics which enable it to be a good replacement of the Observer design pattern. Reactive programming does not use inversion of control. Instead, the developer explicitly defines the control flow through the application. In addition, reactive programming separates the propagation logic and the application logic. Moreover, it enables the developer to implement most of the application logic without side effects. Also, reactive programming can provide certain guarantees concerning the execution order. Lastly, it offers composable reactive values.

Performance Issues

There is some evidence in literature that reactive programming improves the design of reactive applications [37] and developers' code comprehension [36]. However, the propagation of changes in reactive programming is significantly

slower than in the Observer design pattern. Benchmarks have shown that the performance of applications written in reactive programming languages are slower than the Observer design pattern by more than a factor of 1.4 [30]. However, these numbers are very optimistic. In addition, some reactive programming libraries are slower than the Observer design pattern by more than a factor of 50 [30]. Burchett et al. [9], show that the static optimization lowering, can sometimes increase the performance of FrTime [12] by almost a factor of 100 and even then it is still significantly slower than a reference implementation that did not use FRP [9]. This performance disadvantage is even more drastic when the compiler starts to inline parts of the Observer design pattern.

In comparison to the Observer design pattern, the performance disadvantage of reactive programming is expected. That is because reactive programming introduces unnecessary indirections in the update code. Besides the inherent complexity of reactive programming, it often comes with additional functionality, which adds further complexity. Amung these functionalities are guarantees concerning the execution order and a glitch-free propagation. In addition, some reactive programming languages provide functionality for error handling in the propagation. In summary, like most abstractions reactive programming comes at the cost of reducing the runtime performance.

All in all, reactive programming is an appealing solution to develop reactive applications but the performance penalty it introduces is a serious issue. Several efforts to address this problem have been made. For example, researchers proposed to increase the performance by reducing the expressive power of reactive programming libraries [42]. Lowering, which describes a static optimization, attempts to reduce the number of nodes in the dependency graph by merging nodes [9]. In addition, optimizations have been introduced for specific cases like reactive collections [31].

In this thesis we propose a new way to increase the performance of reactive programming languages via just-in-time (JIT) compilation. It is a known technique to speed up interpreted languages. Java is one prominent example which heavily uses JIT compilation to compile on the fly native code portions of Java byte code that are frequently executed. In recent years JIT compilation also had huge success to speed up dynamically-typed languages. An example for this is JavaScript. Both the Mozilla SpiderMonkey JIT and the Crankshaft JIT compiler of the V8 engine greatly increase the performance of JavaScript. For that, both JIT compilers also provide, besides classical optimization techniques, speculative optimizations [24, 11]. Graal [17] is another JIT compiler which demonstrated the effectiveness of JIT compilation to speed up dynamically-typed languages. JIT compilation provides a tool to speed up dynamically-typed languages which are not that well-optimizable by static compilers.

In some sense, the state of reactive programming languages reminds us of dynamic languages like JavaScript a few years ago. The key for their success and spreading has been aggressive virtual machine (VM) optimization. In particular, specialized JIT compilations greatly increased the performance of dynamic languages like JavaScript.

Therefore, we use JIT compilation to optimize reactive programming. In our work we use the Graal JIT compiler because it is designed to provide JIT compilation not only for one specific language but for a wide range of languages. Furthermore, it allows highly speculative optimization. Additionally, it offers with Truffle [44] a language framework which helps to develop languages that provide JIT compilation. In Truffle the developer implements the languages as an abstract syntax tree (AST) interpreter. In addition, the developer can provide optimization and he can influence the JIT compilation of Graal.

Truffle Ruby [7] is an implementation of the Ruby language which uses Truffle as an interpreter and Graal as a VM and a JIT compiler. Ruby is a dynamically-typed, object-oriented general-purpose programming language. Ruby also supports functional programming as well as meta programming. Truffle Ruby was initially developed by Chris Seaton [39] during an internship at Oracle Labs. It is published under an open source license and is now available for researchers. As it is part of the JRuby repository, developers can activate it as an experimental feature. In May 2015, Truffle Ruby supported over 90% of the core Ruby language and approximately 70% of the core libraries [22, 21, 23]. It is currently actively maintained and we expect that it will to be further developed in the near future [2].

Contribution of the Thesis

In this thesis we apply JIT compilation to speed up the execution of reactive programming languages. We use the Graal JIT compiler and the Truffle language framework to implement Reactive Ruby a reactive programming language based on Truffle Ruby. In addition, we provide optimization for certain structures in the dependency graph and for the propagation algorithm. Furthermore, Reactive Ruby offers optimizations for reactive operators.

The goal is to specialize the reactive programming language to achieve a performance which is similar to an implementation based on the Observer design pattern. In a number of benchmarks, Reactive Ruby achieves the same runtime performance as the Observer design pattern, still considering a version of it amenable for JIT compilation.

The contributions of this thesis are the following

- We implemented the reactive programming language Reactive Ruby. It extends Truffle Ruby and it is implemented inside the Truffle Ruby interpreter. To the best of our knowledge, it is the first work which attempts to improve the performance of reactive programming by customizing the JIT compilation.

- We evaluate the performance of Reactive Ruby with JIT compilation support. The evaluation shows that Reactive Ruby with JIT compilation performs well compared to other reactive programming implementations. Reactive Ruby successfully inlines parts of the propagation code and performs map fusion. In benchmarks, the performance of Reactive Ruby applications is in the same area as reference applications which are implemented using the Observer design pattern. In summary, Reactive Ruby is able to reduce the performance gap between reactive programming and the Observer design pattern.

Thesis Structure

The other parts of the thesis are structured as follows:

- Chapter 2 describes (functional) reactive programming. It provides an overview of what reactive programming is, describes some characteristics which can be used to classify reactive programming languages and summarizes some important reactive programming languages.
- Chapter 3 describes Truffle and Graal. For that, it illustrates the Graal VM and the Truffle language implementation framework.
- Chapter 4 describes the Reactive Ruby language as well as the runtime system. The chapters present the design of the language Reactive Ruby, the propagation algorithm and the implementation of Reactive Ruby.
- Chapter 5 evaluates the performance of Reactive Ruby. The chapter explains the design of the benchmarks and describes the evaluation of the peak performance of Reactive Ruby, the impact of the JIT compiler and the warm-up chraceristics of Reactive Ruby.
- Chapter 6 summarizes the central aspects of the work. In addition, the chapter gives an overview over the existing limitations and some possible future work.

2 (Functional) Reactive Programming

This chapter explains reactive programming. As it is still a new paradigm, different reactive programming languages can still differ to a great extent. Therefore, this chapter provides a general description of reactive programming and explains some major differences. In addition, it provides some definitions which this thesis uses. However, it does not give an exhaustive overview over every last detail of reactive programming.

2.1 Introduction

Reactive programming provides the developer with abstractions to develop reactive applications in a declarative style. For that reactive programming languages automatically handle how changes are propagated through the applications: the developer only needs to define what happens when a value changes. Consider the example in listing 1.

```
a = 1
b = 2
c = a + b
a = 2
```

Listing 1: A program to demonstrate the difference between reactive programming and non-reactive programming.

In a traditional programming language like Java, the value of `c` is 3 at the end of the execution. However, in a reactive programming language the value of `c` is 4. When the last instruction assigns the value 2 to `a` the reactive programming language will automatically recompute the value of `c` since it depends on the variables `a` and `b`.

In the example (listing 1), the developer defines that `c` should be the sum of `a` and `b`. He does not need to write the code to ensure that. The language ensures that `c` is always the sum of `a` and `b` for the developer. This has the advantage of not having to write the propagation logic, for the developer. To conclude the example one can argue that the code is more legible and the reactive code is written declaratively.

The general structure of the chapter is inspirited by a paper of Bainomugisha et al. [8].

Events and Behaviors

Reactive programming languages provide (reactive) values and (reactive) operators. The reactive values are events and behaviors.

Events
: An event is a value that represents an infinite stream of changes. This value emits change events which can carry a value every time a change happens. In contrast to a behavior (explained below) the event steam does not hold a value.

Behavior
: A behavior represents a value which changes over time [19]. In general it represents a functional dependency (the *behavior expression*) over other behaviors. Its value is obtained by evaluating the behavior expression. Some reactive programming languages use the name signal instead of the name behavior.

Most reactive programming languages offer events and behaviors. A reason why some languages only offer behaviors is that they can be implemented as a generalization of events [42, 14]. In these languages a developer can consider a behavior as an event that holds a value.

When describing reactive programming most concepts work in the same way for behaviors and events. Furthermore, languages which support both usually provide operators to change an event into a behavior and the other way round. Therefore, we will use the term behavior in this thesis to describe behaviors and events.

Reactive Operators

Reactive programming language operators work on events and behaviors. These operators can transform or combine behaviors into a new behavior. Which operators a reactive programming language provides, differs between languages. However, there is usually an operator to transform a behavior (`map`), an operator to combine several behaviors into one behavior (`merge`) and an operator which works with a state (`fold`). As they are representatives of most reactive programming languages, we explain them here.

`map`
: The `map` operator applies a function `f` to a behavior `b`. In this process it creates a new behavior and the value of this new behavior is the function `f` applied to the value of behavior `b`.

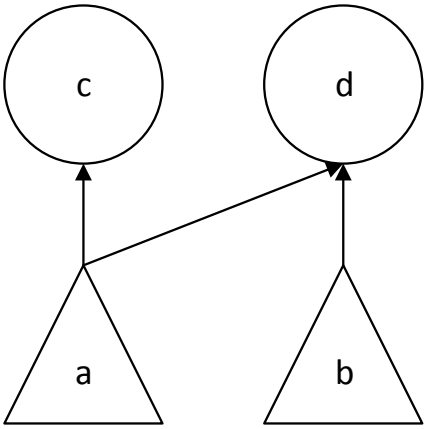

Figure 1: The dependency graph for the Reactive Ruby code in listing 2

`merge`
: The `merge` operator combines two behaviors into a new behavior. The value of the new behavior is the value of the behavior which most recently changed.

`fold`
: The `fold` operator on the behavior `b` takes a function `f` and a value `v`. It creates a new behavior b_{new}. The initial value of b_{new} is `v`. Whenever the value of `b` changes, the value of b_{new} changes to `f(old value of` b_{new}`, value of b)`. In an informal description, the value of b_{new} is `f` applied to the old value of b_{new} and the value of `b`.

2.2 Dependency Graph

A reactive programming language needs to handle dependencies between behaviors. This is necessary since the runtime system needs to update dependent behaviors when the value of a behavior changes. In addition, behaviors represent functional dependencies over other behaviors and form a dependency graph by themselves. On the one hand this graph is a useful tool to visualize the structure of a reactive application. On the other hand it is central data structure used by many reactive programming languages.

In the dependency graph, nodes represent behaviors. The edges between nodes represent dependencies. An edge that goes from a node n_i to a node n_j means that when the behavior n_i changes, the reactive programming language needs to recalculate the behavior n_j. Some terminology which is used throughout this thesis for this structure is: An edge from a node n_i to a node n_j, means that behavior n_j *depends on* the behavior n_i. Furthermore, node n_i is a *predecessor* of node n_j and node n_j is a *successor* of node n_i.

Figure 1 shows the dependency graph for the small Reactive Ruby code in listing 2. Node `c` depends on node `a`. Node `d` depends on nodes `a` and `b`. When behavior `a` changes, behaviors `c` and `d` need to recalculate their values. If behavior `b` changes, behavior `d` needs to recalculate its value. In this dependency graph a triangle stands for a behavior source and a circle stands for a non-source behavior.

```
a = source(1)
b = source(2)
c = a.map { |x| x * 2}
d = a.merge(b)
```

Listing 2: Reactive Ruby code which creates the dependency graph in figure 1.

2.3 Evaluation Model

In reactive programming languages there are two common ways to propagate changes. Changes can be propagated in a push-based fashion or in a pull-based fashion [20].

Push-based evaluation
: In this evaluation model changes are pushed through the dependency graph. Whenever a leaf behavior changes, it notifies dependent behaviors. This results in an update of the reachable closure [15]. This evaluation model is in a way a data-driven evaluation model.

Pull-based evaluation
: In this evaluation model, a behavior pulls the data from its predecessor nodes when it recomputes its value. This process can trigger a recomputation in its predecessor node. This evaluation model is demand-driven.

Some languages use a combination of the push and pull model. However, most reactive programming languages use the push model. This applies in particular to reactive programming languages that extend an imperative / object-orientated language. Besides, if a reactive programming language uses a push-based or pull-based evaluation model, it is possible to distinguish between a lazy and an eager evaluation.

Lazy evaluation
: A reactive programming language that applies lazy evaluation only evaluates the behavior expression, when the value of the behavior is needed. Concerning the code in listing 3 this means that if `a.emit(2)` changes the value of behavior `a` to 2, the value of behavior `c` is updated to 2. The value of behavior `b` is not updated. This is because the value of behavior `b` is not used whereas the value behavior `c` is used. Whenever `c` changes, the new value gets printed, therefore it is used.

Eager evaluation
: A reactive programming language that applies eager evaluation always executes the behavior expression when a predecessor behavior changes. In listing 3, when the value of `a` changes to 2, the values of behaviors `b` and `c` change to 2.

```
a = source(1)
b = a.map {|a| a}
c = a.map {|a| a}
c.onChange {|x| put x }
a.emit(2)
```

Listing 3: Reactive Ruby example to demonstrate the difference between eager and lazy evaluation.

2.4 Glitches and Glitch Freedom

A glitch in a reactive programming language means that a behavior has a temporary inconsistent value during a propagation turn. This can happen if a behavior executes its expression before all predecessor behaviors are updated.

We consider the reactive application in listing 4 and the dependency graph for this application in figure 2. In this example, behaviors `b` and `c` should always have the value of behavior `a`. The value of behavior `d` should always be twice the value of behavior `a`.

A glitch can happen in the following way:

1. The instruction `a.emit(2)` changes the value of the behavior `a` to 2. Then behavior `a` notifies behavior `b`.

2. Behavior `b` recalculates its value. The value of the behavior `b` is now 2. Behavior `b` then notifies behavior `d` that it has changed.

3. Behavior `d` recalculates its value which gets printed. At this point, the value of behavior `b` is 2, but the value of behavior `c` is still 1. Therefore, the new value of `d` is 3. Behavior `d` is in a temporarily wrong state which means that a glitch has occurred.

4. Behavior `a` notifies behavior `c` that it has changed.

5. Behavior `c` recalculates its value. The new value of behavior `c` is 2. Behavior `c` notifies behavior `d` that it has changed.

6. Behavior `d` recalculates its value again. Hereafter, behavior `d` has the correct value of 4. The correct value is printed.

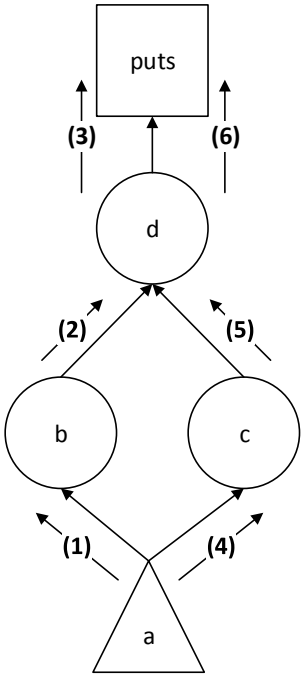

Figure 2: A dependency graph with a propagation order that causes a glitch

```
a = source(1)
b = a.map {|a| a}
c = a.map {|a| a}
d = b.map(c) {|b,c| b + c}
d.onChange {|x| puts x}
a.emit(2)
```

Listing 4: Reactive code which demonstrates how a glitch can occur.

Not all reactive programming languages are glitch-free. In a local setting, a common way to achieve glitch freedom is a propagation algorithm that uses a topologically ordered dependency graph [8]. Glitch freedom in a distributed setting is more complex, but also possible [15].

2.5 Static and Dynamic Dependencies

Reactive programming languages can have dynamic or static dependencies or both.

Static Dependencies

A reactive programming language has static dependencies if existing dependencies do not change during runtime. The only changes in the dependency graph are the insertion of new nodes and the connection of these new nodes with the dependency graph. The language adds new nodes when it creates new behaviors. That means the language adds no new edges between already existing nodes and it does not remove edges from the graph.

The implementation of a glitch-free propagation algorithm for a reactive programming language with static dependencies is simpler than for one with dynamic dependencies. Furthermore, static dependencies allow for optimization which are not possible in a language with dynamic dependencies. A popular reactive programming language that has only static dependencies is ELM [14].

In listing 5, the first part shows language features which create static dependencies. In this example, the operators `map` and `filter` create nodes which incoming edges will never change.

Dynamic Dependencies

In a reactive programming language with dynamic dependencies the structure of the dependency graph can change during runtime. During runtime, instructions can remove and add edges of existing nodes. Higher order reactives or

```
#static
s = source(1)
m = s.map    { |x|  x }
f = m.filter{ |x|  x > 2 }

#dynamic
s1 = source(1)
s2 = source(2)
bs = source(true)
expr = behavior {
        if(bs)
                s1
        else
                s2
}
```

Listing 5: A reactive programming application. The first part shows reactive operators which create static dependecies. The second part creates dynamic dependencies.

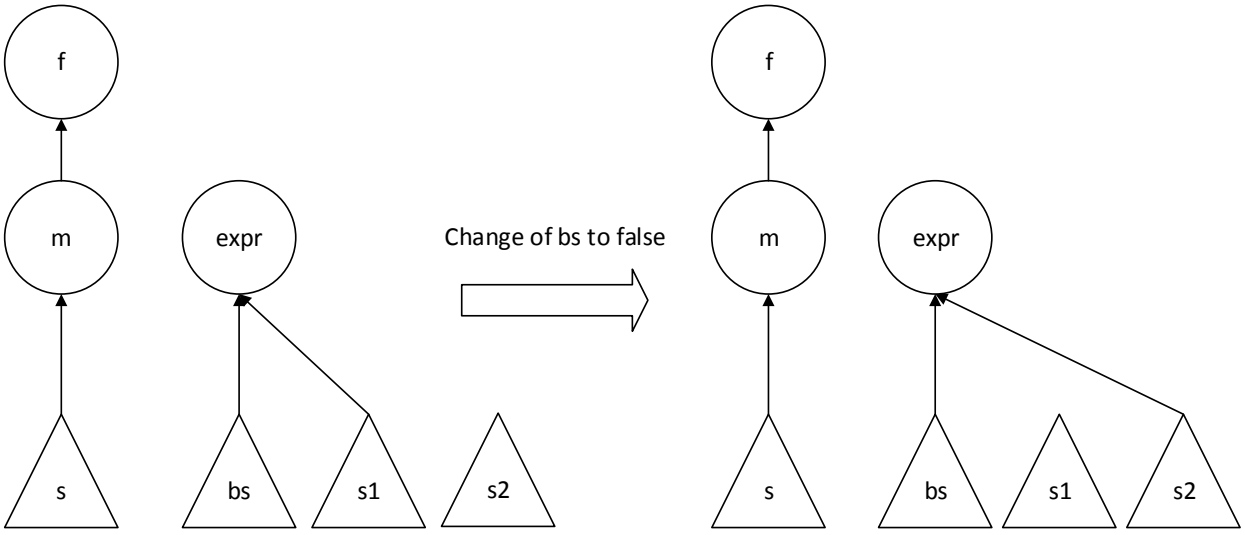

Figure 3: Two dependency graphs: the left one is static and the right one is dynamic.

behavior expressions with dynamic dependency discovery introduce dynamic changes. For example, Flapjax [34] is a reactive programming language with dynamic dependencies.

Let us consider the conditional behavior expression `expr` in listing 5. The node `expr` initially depends on `bs` and `s1`. When the value of `bs` changes to false, `expr` depends on `bs` and `s2`. Therefore, the edge from `s1` to `expr` is removed and the edge from `s2` to `expr` is added. Figure 3 visualizes the change of the dependencies of node `expr`.

2.6 Lifting

Lifting is the technique of applying a function which is defined for non-reactive values to a behavior. For example, the function + does not work on behaviors even if the behaviors contain numbers as their values. To achieve that, the function + must be lifted. Bainomugisha et al. [8] classify lifting into implicit lifting, explicit lifting and manual lifting.

Implicit lifting

Implicit lifting could also be called automatic lifting. A language that applies implicit lifting lifts a function which cannot work on behaviors if it is applied to a behavior. Now it can operate on these behaviors.

Explicit lifting

A language that applies explicit lifting provides the developer with operators to lift functions. The developer can

use these operators to apply non-reactive functions to behaviors. Moreover, languages which use explicit lifting often provide a set of standard functions which are already lifted [8].

Manual lifting
: In a language that uses manual lifting, the developer needs to lift a function manually for working with behaviors. The difference to explicit lifting is that the language does not provide operators to lift functions. Instead, the developer needs to access the current value of a behavior manually. Then he can apply a function to that value.

2.7 Existing Languages

In the following, we describe some popular (functional) reactive programming languages and summarize some important properties of them.

Fran

Fran [19] was developed to simplify the programming of GUI applications. Elliott Conal, Fran's developer, has given the first description of FRP. Elliott introduced *behaviors* and *events*. Fran's initial version uses a pull-driven evaluation model and the user needs to perform explicit lifting. Unfortunately, the initial implementation suffers from space and time leaks. This means that the memory can grow unexpectedly (space leak) and the computing time can unexpectedly become arbitrarily long (time leak). The language received an update [20] and now applies a push and pull evaluation model.

ELM

ElM is a FRP language which targets the development of GUIs for web applications [14]. ELM is a purely functional language and only offers signals. Its syntax guarantees that no higher order signals are created [14]. Therefore, its has static dependencies. For the propagation of changes it uses a push-based evaluation. This allows treating signals in ELM like events.

ELM introduced asynchronous signals, which allow asynchronous computation of signals [14]. ELM handles asynchronous signals in the way that they are represented by two nodes in the dependency graph. The first node represents a sink in the dependency graph. This node depends on all nodes the asynchronous signal depends on. The second node is a source node in the dependency graph which emits a change event when the asynchronous signal finishes its reevaluation. All nodes that depend on the asynchronous signal depend on the source node. In summary, using this technique the propagation algorithm does not need to have any special knowledge about asynchronous nodes.

ELM uses a global event dispatcher for the event propagation. When the dispatcher starts the event propagation, ELM evaluates its code in two stages. In the first stage, ELM evaluates the non-reactive code. In the second stage, it evaluates signals in a push-based fashion. Like many other FRP languages, ELM uses a direct acyclic graph for the propagation. There is a difference to most other propagation algorithms. When a source node changes, ELM sends a change event while at the same time all other source nodes send a noChange event.

Scala.React

Scala.React [30] is a reactive programming language and supports events and signals which are first-class values in Scala.React. It is implemented as a Scala library.

Besides the typical composition functions like `merge` and `filter`, Scala.React also offers a data flow domain-specific language (DSL). This DSL allows the developer to build a state machine over events.

Scala.React has a glitch-free propagation algorithm which uses a push-based evaluation model as it is common for reactive programming languages which extend object-oriented languages. A central manger controls the propagation algorithm which uses a topologically ordered dependency graph to guarantee its glitch-freedom. In contrast to other propagation algorithms, Scala.React distinguishes between strict and lazy nodes. During propagation, a strict node is updated even if nobody is interested in the node. A lazy node is only updated if someone is interested in that node. In other words, it applies a mix between eager and lazy evaluation. In addition, the propagation algorithm can handle dynamic changes in the dependency graph. This is important since Scala.React allows creating signals via `signal expressions` which use dynamic dependency discovery. Therefore, its dependency graph can change dynamically e.g. if a signal uses the expression `{a() then b() else c()}`.

Flapjax

Flapjax [34] is a FRP language which compiles to JavaScript. Alternatively, Flapjax can directly be used as a library in JavaScript.

Flapjax supports event streams and behaviors and both are first-class values. Besides the typical functions like for example `lift` and `fold`, Flapjax provides methods to access DOM elements in a FRP style. In addition, Flapjax supports higher order FRP. Unfortunately, higher order FRP in Flapjax can cause memory leaks [30].

Flapjax has a glitch-free propagation algorithm that can handle dynamic changes in the dependency graph. For that it uses a topologically ordered dependency graph. Moreover, the propagation algorithm uses a push-based evaluation strategy. A central manager performs the propagation.

Rx

Rx [3] is a reactive programming language which was initially developed for .NET by Microsoft. A number of other languages adopted the ideas and concepts behind Rx.NET. Among others, a port of Rx exists for the languages Scala, Java and JavaScript.

Rx extends a language with an `Observable` which represent a data stream and offers beside event like properties, functionality for error handling and ending of event streams.

The propagation algorithm of Rx uses a push-base evaluation. In addition, it supports asynchronous computation. Unfortunately, it does not offer a glitch-free propagation by default.

REScala

REScala [4] is a library which extends Scala with reactive programming. It embraces ideas from classical FRP as well as from event-driven programming [37]. The language offers events and signals. Besides classical operators like `map` REScala offers `signal expressions`.

REScala has a glitch-free propagation algorithm which is evaluated in a push-based fashion. In contrast to other reactive programming languages, it also provides a glitch-free propagation algorithm for a distributed setting [36].

In an empirical study, Salvaneschi et al. provided evidence that it is easier to understand reactive applications written in REScala then reactive applications written in an object-oriented style [36].

Despite covering important related work this list of reactive programming languages is by no means complete. Some further notable mentions will follow below. Real-Time FRP (RT-FRP) [41] is a statically-typed FRP language. It adds guarantees regarding the execution time. For that it uses only a subset of the FRP expressibility.

Event-driven FRP (ED-FRP) [42] is built on the ideas of RT-FRP. ED-FRP is a FRP language which is used to program robots. It also restricts the expressibility of FRP to allow efficient computation. Among others, it does not allow higher order behaviors. ED-FRP is compiled into a subset of the c language and the compiler performs some static optimizations to increase the performance.

3 Graal and Truffle

This chapter provides background information about the Graal JIT compiler and the Truffle language implementation framework [44, 43, 16]. In this chapter the focus is more on Truffle than on Graal as the Reactive Ruby runtime system developed in this thesis does not directly interact with Graal. However, the way Graal works directly influences the performance of a language which is implemented in Truffle.

3.1 Graal

Graal VM extends the Java HotSpot VM with a speculative JIT compiler. Besides providing a basis for language development, Graal VM can be used to develop static analyses. In this thesis we use Graal as a JIT compiler. For that reason, the rest of the chapter only describes Graal features which are important in language development.

The Graal JIT compiler is written in Java. One of the main goals of Graal is to provide an aggressive JIT compiler. In this context, aggressive means that the compiler creates a machine code which is highly specialized. Such optimizations come at the cost of Graal having to deoptimize the JIT compiled code sometimes.

Figure 4 shows the architecture of a system that works with Graal and Truffle. Graal with the Hotspot VM constitutes the first layer above the operation system. The next layer is the guest language VM, which is in this thesis the Reactive Ruby VM. The guest language VM uses the Truffle language implementation framework for its implementation. Applications written in the guest language build the last layer.

The Graal VM and the guest language VM work closely together. As the Graal VM is written in Java, it allows the Truffle language framework an easy access and to control features of the Graal VM. Among others, Truffle and the guest language VM can influence the JIT compiler of Graal. In addition, the guest language VM can define when Graal needs to deoptimized parts of the JIT-compiled code.

This layered approach has the advantage that the guest language VM can use the existing benefits from the Java VM. Among others, the guest language VM can use features like garbage collection, threads and a memory model. For that reason, the guest language developer can focus on the language semantics and does not need to implement all VM features.

3.1.1 Speculative Optimization and Deoptimization

Let us consider the Ruby source code in listing 6 as an example for speculative optimization and deoptimization. This code assigns a value to the local variables `a` and `b`. After that, it adds `a` and `b` together.

In general, when the interpreter working with Graal executes an application it collects profiling information. After collecting this profiling information, Graal uses this information for its speculative optimizations.

Let us assume that the profiling information for method `m` is that every time the interpreter executes `m`, the type of `a` and `b` is integer. This information suggests the speculation that the type of `a` and `b` is always integer. Then the JIT compiler can compile a fast specialized version of `m` in which `a + b` only works for the type integer. In summary, Graal performed a speculative optimization for the method `m`.

There is obviously no guarantee that the speculation of `a` and `b` having the type integer will always hold. In case that the type of `a` or `b` change, e.g., to double, the optimized code of the method `m` is not able to calculate the correct result. In particular, the first two instructions (`a = magicallyGetA()` and `a = magicallyGetB()`) can still work. However, the compiled code e.g., will contain a guard in front of the code of the instruction `a + b`. This guard ensures, that the type is integer and will trigger deoptimization if the type is not integer. In the running example, Graal invalidates the code for `m` and continues the execution of `m` in the interpreter.

Deoptimization is the process of invalidating the optimized compiled code and continuing the execution in the interpreter. This process is complex because JIT compiled code can change the state of an application. In the example, the execution of `magicallyGetA()` or `magicallyGetB()` may change the state of the application (have side effects). Therefore, when Graal switches the execution from the compiled code to the interpreter it cannot just reexecute the method `m` in the interpreter. Instead, it needs to continue the execution in the interpreter after it has read the value of `a` and `b` and before `a + b` fails. For that, Graal needs to reconstruct the program state so that it can continue the execution of method `m` in the interpreter. Therefore, the optimized code saves meta information which allow Graal this reconstruction. The name of these meta information in Graal is frame state. For more information [16].

```
def m()
        a = magicallyGetA()
        b = magicallyGetB()
        return a + b
end
```

Listing 6: Ruby code to demonstrate optimization and deoptimization in Graal.

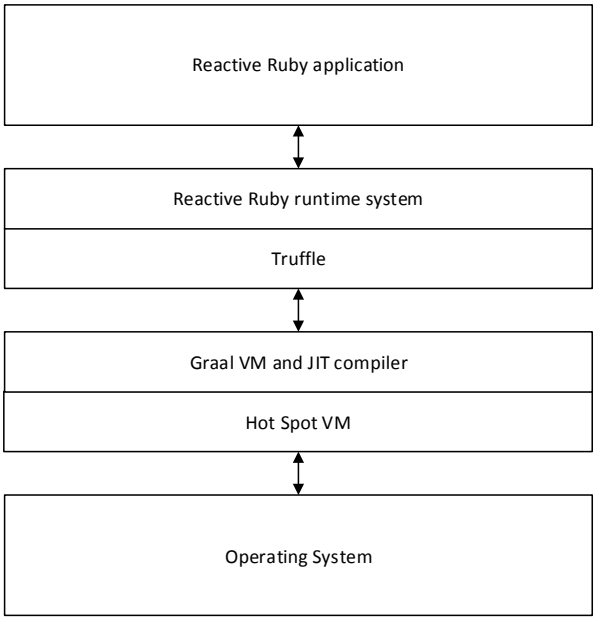

Figure 4: System structure of a runtime system implementation in Truffle which is managed by Graal.

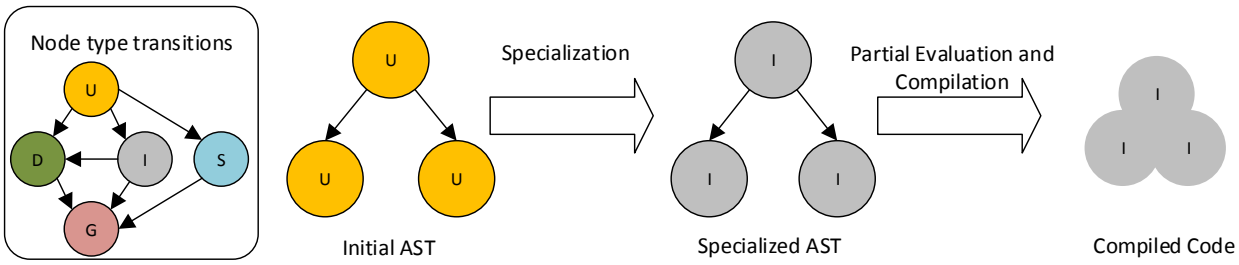

Figure 5: AST specialization for type information and partial evaluation

3.2 Truffle

Truffle is a framework to develop languages which support speculative optimization. It is coupled with Graal and uses the possibilities Graal offers to control the compilation process.

Languages which are implemented with Truffle are implemented as an AST interpreter. AST nodes implement the semantics of the language. Specifically in Truffle several AST represent the application and the methods. Every AST has a root node with an `execution` method, which is the starting point of the interpretation of this AST. Moreover, all nodes have an arbitrary number of nodes as child nodes. Like the root nodes all other nodes have `execution` methods. The AST is interpreted by the execution of the `execution` method which executes the `execution` method on some or all child nodes and so on [44].

In general, AST interpreters are considered to be slow [44]. This is one of the reasons why the Java VM uses a byte code interpreter. However, Truffle uses self-optimization of the AST and partial evaluation to increase the performance of the implemented AST interpreter. Both techniques are explained later.

The AST representing the application is created from the source code. Initially, this AST represents the code in its general form, the AST is uninitialized. That means it does not contain any speculative optimizations, e.g., the AST of a dynamically-typed language does not contain any type information. During interpretation Truffle rewrites the AST, it replaces uninitialized nodes with specialized nodes (self-optimization). When the AST reaches a stable state, Truffle applies partial evaluation to the stable AST. Among others, partial evaluation removes the calls between nodes in the AST.

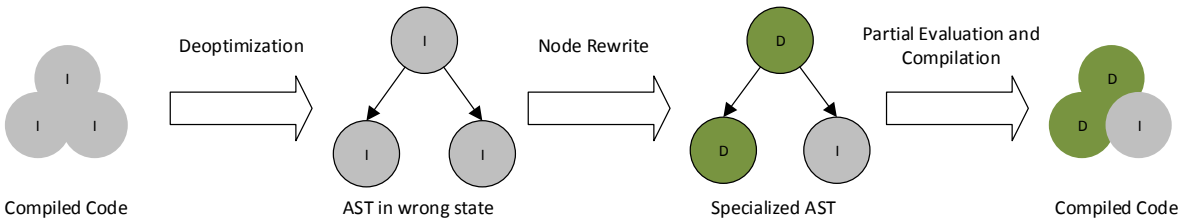

Figure 6: Deoptimization of the compiled code. Rewriting of the wrong AST state. Compilation of the correct AST.

Figure 5, based on [28], gives an overview of how Truffle works. It shows the process of rewriting the AST to a specialized AST based on type information. As an example, the expression `a + b` can create an AST like the one in figure 5. The two nodes in the bottom read the variables `a` and `b` and the top nodes calculate the addition.

The possible types are integer, double, string and generic. The double type includes the integer type and the generic type can handle all types.

In figure 5 the AST starts in the uninitialized state. The `u` in the nodes stands for uninitialized. During the interpretation Truffle specializes the AST based on collected profiler information. Here the AST specializes in an AST where all types are integer. That means the uninitialized nodes that read the values of `a` and `b` are replaced by specialized integer read nodes. In addition, the uninitialized + node is replaced by an integer addition node.

When the AST reaches a stable state, Truffle may trigger the JIT compilation process. An AST is in a stable state if it has not changed for a predefined number of executions. In more detail, the following happens if Truffle triggers the JIT compilation process: First, Truffle performs optimizations on the AST and creates an intermediate representation. For that, Truffle performs partial evaluation on the AST which is explained in more detail in section 3.2.5. Most notable, partial evaluation removes the calls between nodes in the AST. After that, Truffle hands the IR of the optimized AST to Graal. Then Graal performs optimization on the IR and JIT compiles it. In summary, Truffle performs optimizations on the AST level and creates an IR. Then Graal performs optimization on the IR and JIT compilation.

In figure 5, Graal JIT compiled the expression `a + b` with the speculation that the type of `a` and `b` is integer. This code is efficient, however, it cannot handle other types. Graal triggers deoptimization when e.g., the type of `b` becomes double.

Figure 6 visualizes the process of deoptimization and afterwards the JIT compilation of generalized AST. In detail, first the deoptimization of the compiled code happens, e.g., because the guard which ensures that `b` is integer fails. After that, Truffle continues the execution in the AST. At this point the AST is still in a specialized state, all nodes are specialized for the type integer. Since the speculation that `b` is integer is wrong, Truffle rewrites the nodes `b` and + into a more general node which can handle numbers of type double (the double node can handle integers). After this generalization, when the AST again reaches a stable state, Truffle may trigger JIT compilation.

The process of rewriting the AST, based on collected runtime information, into a more specialized AST is named *self-optimization*.

3.2.1 Node Rewriting

During the interpretation a node can replace itself and its children with new nodes. These node replacements allow a local specialization and generalization of an AST. For example a node that represents an addition can specialize into an integer addition node. Later, the integer addition node could be generalized into a double addition node.

Nodes that support specialization have several states which are implemented in different nodes. These nodes are an *uninitialized* node, multiple *specialized* nodes and the *generic* node. The box *node type transitions* in figure 5 shows these nodes for a node that performs type specializations, e.g., the + node. This node has an uninitialized node, three specialized nodes for the type integer, double and string and the generic node. These three types of nodes which together implement the node semantics and its local optimizations do the following: The uninitialized node is the state a node has had before it was executed the first time. When the uninitialized node is executed, it is rewritten into a specialized node that can handle the input (node specialization). If no specialized node can handle the input, it is rewritten into the generic node. The specialized nodes provide a fast implementation for a subset of the node's semantics. When a specialized node is executed, it handles the input if possible. If it cannot handle the input, it is rewritten into another more general specialized node or into the generic node (node generalization). The generic node provides an implementation of the node that can handle all valid inputs.

For the node rewrites the developer needs to ensure the following properties.

Completeness
Specialized nodes may only work on a subset of the generic nodes semantics. However, they must provide rewrites to a generic node which can handle the whole semantics, or they must be able to redirect the execution to a node, which can handle the whole semantics.

Finiteness
The number of times a node is rewritten must be finite. That means that cycles in the node replacement are not allowed. At some point, the node must be rewritten into the generic node which can handle the whole semantics.

Locality
Node rewritings are only allowed locally. That means when a node is rewritten, the rewriting may only affect this node and its child nodes.

3.2.2 Inlining and Tree Cloning

A problem with collecting profiling information at runtime is pollution of feedback. This can happen when several call sites call a method with different inputs [44]. A specialized method needs to be general enough to work for all call sites. For example, the non-primitive type specialization of a method in Java can suffer if different call sites call this method [45].

To reduce the effect of pollution of feedback, Truffle applies a technique that is called tree cloning [44]. In Truffle, all nodes, and therefore also the tree which represents a method, can clone themselves. If a method call site is hot, Truffle can clone the method which the call site frequently calls. Then the hot call site works with a clone of the method. When Truffle clones a method the AST of the cloned method is set into its initial state. That technique allows the specialization of the method for a certain call site. When Truffle performs inlining at that call site, it inlines the cloned method.

The cloning and inlining in Truffle work at the AST level. This means that when Truffle performs inlining, it copies the whole tree of the method into the call site. The cloning and inlining happens before Graal does further optimizations. The advantage is that Graal is able to apply optimizations to one method which contains the inlined methods instead of several small methods. Therefore, Graal can apply certain optimization techniques to the method of the call site as well as to all inlined methods, even if these techniques would normally only work for the method of the call site.

The cloning and copying of methods increases the size of the AST. Therefore, the Truffle framework uses heuristics to detect, where it is beneficial to clone a method. The language developer can control if Truffle should never/sometimes/always clone methods. For example, the language designer could define that Truffle always clones and inlines the `foreach` method of a collection library.

3.2.3 Assumptions

As we already discussed, deoptimization is the process of invalidating compiled code and resuming the execution in the interpreter. Graal and Truffle allow two ways to trigger deoptimization. The compiled code can explicitly trigger deoptimization by a call to the deoptimization method. The other way to trigger deoptimization are assumptions.

An assumption in Truffle is a global flag. Initially, the flag is valid and it can be set to invalid exactly once. Nodes in the AST can reference an assumption and check if this assumption is valid. When an assumption is set to invalid, compiled code which depends on that assumption becomes invalid. Therefore, Graal triggers deoptimization the next time it executes the compiled code.

This feature is important, for example, for the implementation of a function cache in a language which allows a function redefinition. In this example, assumptions provide a way, to invalidate the compiled code when the cached function changes. Therefore, the compiled code does not need to check, if the cached function is valid every time it is executed. This approach increases the performance, since usually a function does not get redefined that often.

3.2.4 Object Storage Model

Truffle provides an object storage model (OSM)[43]. For a more detailed explanation see the original Truffle OSM paper [43].

The OSM can be used to implement objects in Truffle. In more detail, the OSM provides space where the object can save primitives values and object references. In addition, it provides the functionality to allocate more space for attributes if necessary. Furthermore, the OSM manages the mapping from attribute names to locations where the attributes are saved. Moreover, it provides the functionally for a fast read and write access to an attribute.

3.2.5 Partial Evaluation

Truffle applies partial evaluation to an AST when that tree reaches a stable state. This ia a condition that holds when Truffle has executed the AST for a number of times without a change.

The partial evaluation assumes that the AST will not change. This allows replacing the virtual calls with direct calls which call the child nodes in the `execute` methods . After that, the partial evaluation inlines all these direct calls. This process creates a unit which represents the stable AST. In other words, the process removes the edges in the AST.

Another advantage of the assumption of the AST being stable is that many values in the AST become constant [44]. This allows replacing a variable with an actual value during partial evaluation.

In general, an AST contains nodes which trigger node replacement. Obviously, rewriting is not allowed since partial evaluation assumes a stable AST. Therefore, during partial evaluation all rewrite instructions are replaced by deoptimization points.

The result of the partial evaluation is an intermediate representation (IR). This IR represents the semantics of the AST. Then Truffle gives the IR to Graal. After that, Graal performs further optimization and compiles the IR.

4 Reactive Ruby

This chapter describes Reactive Ruby and its implementation. We initially present a brief overview of Reactive Ruby in a nutshell, then we describe the propagation algorithm. After that, we describe the implementation of Reactive Ruby and some optimizations.

4.1 Design of Reactive Ruby

Reactive Ruby extends the Truffle Ruby language with abstractions for reactive programming. Reactive Ruby provides behaviors as well as the typical reactive programming operators on behaviors.

The reactive programming part of Reactive Ruby is integrated into Ruby's object-oriented programming model. Behaviors are first-class values, can be passed as parameters and returned by functions. Objects can store behaviors in fields. Also, Reactive Ruby supports the creation of behaviors at runtime.

4.1.1 Reactive Ruby in a Nutshell

We present the design of Reactive Ruby making use of some examples. The example below prints the current time every second. For that, `timeB(1)` provides a behavior that represents the current time, which gets updated every second. In addition, we need to register a method to the behavior which prints the current time whenever the behavior changes.

```
time = timeB(1)
time.onChange { |x| puts x}
```

Lets us consider another typical reactive example, namely printing the current position of the mouse. Here, `mouseB()` provides a behavior which represents the mouse. This behavior contains more information than the position of the mouse. Therefore, `mouse.map { |m| [m.xpos, m.ypos] }` creates a new behavior which contains the position of the mouse, which gets printed whenever it changes.

```
mouse = mouseB()
mousePos = mouse.map { |m| [m.xpos, m.ypos] }
mousePos.onChange { |pos| puts pos}
```

The next example adds up all even numbers. In this example, method `rangeB(1,100)` creates a behavior which initially holds the value 1 and then stepwise increases until it holds the value 100. The `range.filter` creates behavior `even` which holds the last even value of `range`. Then `range.fold` produces behavior `sum` that adds up over all values of the behavior `even`. After that, `range.map` combines behaviors `range` and behavior `sum`. The value of this behavior is printed whenever it changes.

```
range = rangeB(1,100)

even = range.filter(0) { |x|
        x.to_i % 2 == 0
}

sum = even.fold( 0 ) { |acc, val|
        acc + val
}

combine = range.map(sum) { |x, y| [x,y] }

combine.onChange {|x| puts "sum: #{x}"}
```

The following example examines if the user inputs for a mailing list registration form are valid. It checks if the user provides a valid email address and it ensures that the user selects at least one mailing list. Lastly, it makes sure that the user selects between immediate notifications or daily summaries.

In this example, the code initially creates the behaviors which represent different form elements. Behavior `mail` represents the provided email address and `selectableLists` is an array of behaviors which represent the selectable mailing lists. In addition, `immediatUpdates` and `dailyUpdates` are behaviors which contain the information if the user selected immediate or daily updates. After that, the code checks if the user provides a valid input. For that, the code creates the behaviors `checkEmail`, `checkListSelected` and `checkUpdate`, which are true if their representative input is valid. Behavior `checkEmail` uses the method `checkMail` to validate the email address. The next behavior `checkListSelected` needs to check that the user has selected at least one mailing list. To manage that, it uses the `atLestOneListSelctedB`

helper method. This method creates a new behavior which combines the behaviors in `lists` and makes sure that at least one of them represents the value true. The last behavior `checkUpdate` combines the behaviors `immediateUpdates` and `dailyUpdates` and checks that the user selected exactly one behavior.

The last line in the code calls the method `visualizeChecks` which visualizes the checks. This method, e.g., displays a green OK symbol when the check is valid, and a red cross if it is not valid besides the email address.

```
#helper method
def atLeastOneListSelctedB(lists)
        return lists.first.map(*lists.drop(1)) { |*args|
                #args represents the current value of all behaviors in lists
                #inject is Rubys fold method
                args.inject { |acc, x| acc || x }
        }
end

#input behaviors
mail = textfield("inputEMail")
selectableLists = getSelectableMailingList()
immediateUpdates = check("imUpdate")
dailyUpdates = check("dayUpdate")


#check
checkEmail = mail.map {|m| checkMail(m)}
checkListSelected = atLeastOneListSelctedB(selectableLists)
checkUpdate = immediatUpdeas.map(dailyUpdates) {| imUp, dayUp| imUp ^ dayUp}


#visualize
visualizeChecks(checkEmail,checkListSelected,checkUpdate)
```

4.1.2 Behaviors and Sources

Reactive Ruby's reactive values include behaviors to express functional dependencies and source behaviors (i.e., sources) to trigger changes in the reactive system.

Source

The source behavior is the starting point of a reactive computation. A source behavior wraps a normal Ruby value. Then, a developer can use this source to create behaviors which will be updated whenever this source changes. In addition, the imperative code can change the value of a source behavior.

Behavior

A behavior describes a functional dependency, the *behavior expression*, among other behaviors. It represents a value which changes over time. For convenience, we use the value of a behavior for the value which is currently represented by the behavior. The developer can register handlers on the behavior which notify the imperative code of any changes. The imperative code can read the value which the behavior represents. However, it cannot change the value of a behavior.

The following example demonstrates sources and behaviors. The code snippet initially creates two sources `a` and `b` which represent the values 1 and 2. After that, it creates a behavior `c` which adds the behaviors `a` and `b` together. The operator `emit` changes the value of source `a` from imperative code. As behavior `c` is functionally dependent on `a` and `b`, it is updated to 7. The imperative code cannot change the value of `c`.

```
a = source(1)
b = source(2)
c = a.map(b) { |a,b| a + b }
a.emit(5)       # change the value of a imperatively
# value of c is 7
# c.emit(1) creates a runtime error
```

4.1.3 Operators

Reactive Ruby provides operators to transform a behavior, to combine behaviors and to use the past values of a behavior.

`map() {|val| ... } -> Behavior`

The `map` operator on a behavior `b` consumes a Ruby block `f` (`{|val| ... }`). It creates a new behavior b_{new}, which represents `f` applied to the value of `b`.

The example below applies the `map` operator to the behavior `source`. It creates the new behavior `twice`. The value of behavior `twice` is always twice the value of `source`.

```
source = source(1)
twice = source.map { |x| x * 2 }
source.emit(2) # -> twice.now = 4
source.emit(4) # -> twice.now = 8
```

`map(b1,...,bn) {|v0,v1,...,vn| ...} -> Behavior`

This `map` operator is a generalization of the previous case. The `map` operator on a behavior `b` takes an arbitrary number of behaviors b_1, ..., b_n and a Ruby block `f` (`{|v0,v1,...,vn| ...}`) as arguments. It creates a new behavior b_{new} by applying `f` to the values carried by `b` and b_1, ..., b_n.

In the example, below the `map` operator combines behaviors `fname` and `sname` in the new behavior `fullName`. This behavior represents the functional dependency `{ |vn, nn| nn + " " + vn }` where `vn` and `nn` are the values of the behaviors `fname` and `sname`. The block creates a string by concatenating the values of the behaviors `fname` and `sname`, which is initially Hans Peter. In other words, behavior `fullName` represents the full name if `fname` represents the first name and `sname` represent the surname.

```
fname = source("Hans")
sname = source("Peter")
fullName = vname.map(nname) { |vn, nn| nn + "␣" + vn }
puts fullName.now # Hans Peter
sname.emit("Maier")
puts fullName.now # Hans Maier
```

`fold(init) {|acc,v| ... } -> Behavior`

The `fold` operator on a behavior `b` takes a Ruby value `init` and a Ruby block `f` (`{|acc,v| ... }`) as its arguments. It creates a behavior b_{new}, which depends on itself and the behavior `b`. Initial b_{new} represents the Ruby value `init`. Whenever behavior `b` changes, `fold` applies `f` to the current value of b_{new} and the value of `b`.

In the example below, the `fold` operator creates the sum over all values of the `number` behavior.

```
number = source(0)
sum = number.fold(0) { |oldValue, num| oldValue + num }
puts sum.now # 0
number.emit(1) # sum.now = 1
number.emit(4) # sum.now = 5
```

`filter(init) {|v| ... } -> Behavior`

The `filter` operator takes as arguments a Ruby value `init` and a Ruby block `f` (`{|v| ... }`). It creates a behavior b_{new} which filters out certain values of behavior `b`. Initially, the value of behavior b_{new} is `init`. Whenever `b` changes and the function `f` evaluates to true for the value of `b`, the value of b_{new} gets updated to the value of `b`.

`merge(b1,...,bn) -> Behavior`

The `merge` operator on a behavior `b` takes an arbitrary number of behaviors b_1, ..., b_n as arguments. It creates a new behavior b_{new}. The value of the new behavior b_{new} is initially the value of the behavior `b`. After the initialization, the value of the behavior b_{new} is the value of the behavior which changed most recently. In the case that during a propagation turn more than one predecessor behavior changes, the `merge` operator selects the value of the behavior that has the left-most position in the argument list and also changes its value in this turn.

`take(n) -> Behavior`

The `take` operator on a behavior `b` takes a number `n` as a parameter. It creates a new behavior b_{new}. For the next `n` changes of `b` the value of b_{new} is the value of `b`. After that, the value of b_{new} does not change anymore.

`skip(n,v) -> Behavior`

The `skip` operator on a behavior `b` takes a number `n` and an initial value `v` as parameters. It creates a new behavior b_{new}. The value of b_{new} is initially `v`. After the behavior `b` changed `n` times the value of b_{new} is always equal to the value of `b`.

`sampleOn(b1) -> Behavior`

The `sampleOn` operator on a behavior `b` takes another behavior b_1 as a parameter. It creates a new behavior b_{new}. Whenever the value of `b` changes, the behavior b_{new} changes its value to the value of b_1. When the value of b_1 changes, the value of b_{new} does not change.

4.1.4 Behavior Expression

`Behavior expressions` are defined through the syntax `behavior {behaviorexpr}`, where `behaviorexpr` is a side effect free Ruby block. A `behavior expression` creates a new behavior whose value depends on other behaviors via `behaviorexpr`. That means that `behaviorexpr` is recomputed if a behavior that appears in the expression changes its value. In the `behaviorexpr`, the value of a behavior is accessed via the operator `value`.

In the example below the `behavior expression` creates a behavior `c` whose value is the addition of `a` and `b`. The behavior `c` accesses the values of `a` and `b` inside the `behaviorexpr` via the operator `value`.

```
a = source(0)
b = source(1)
c = behavior {
        a.value + b.value
}
```

A `behavior expression` creates a behavior which statically depends on all behaviors in `behaviorexpr`. Section 4.7.1 explains how the system calculates the dependencies from the `behaviorexpr` and some limitations.

The following example shows a consequence of the static dependency discovery. In this example behavior `d` depends on the behaviors `a`, `b` and `c`. Therefore, the value of `d` is reevaluated when behavior `c` changes. This is an unnecessary reevaluation if the contraflow through the block is considered. Since the value of behavior `a` is true, the interpreter does not execute the else branch.

```
a = source(true)
b = source(0)
c = source(1)
d = behavior {
        if(a.value)
                b.value
        else
                c.value
        end
}
```

4.1.5 Integration with Imperative Code

Besides operators for composing behaviors, Reactive Ruby provides operators that allow the integration of reactive code and imperative code.

`emit(value)`

The `emit` operator changes the value of a behavior source. The execution of the `emit` operator triggers the start of a change propagation turn.

`now()`

The `now` operator returns the current value of a behavior or a source.

`onChange() {|v| ... }`

The `onChange` operator registers a Ruby block on the behavior. This Ruby block will be executed when the value of the behavior changes. This operator is useful since the behavior expression should be side effect free.

`remove(proc)`

The `remove` operator removes the provided `proc` from a behavior if it is registered on the behavior.

4.2 Propagation Algorithm

The runtime system of Reactive Ruby provides JIT compilation for the reactive code. Before it actually JIT compiles code, it performs speculative optimization. The selection of the propagation algorithm highly influences how well Truffle's node rewriting can improve the performance of a reactive programming language. In general, Truffle allows local and global optimizations. However, it accelerates if it can perform aggressive local optimization via node rewriting.

For example, after node rewriting the specialized AST for a chain of behaviors should contain almost no propagation-handling code. In a chain, one behavior needs to notify exactly one other behavior after it changes. In addition, a node does not need to perform a glitch free check. Therefore, after node rewriting the AST for this structure should not contain a glitch-free check. In addition, the specialized node that notifies the next behavior in the chain should contain a direct call to the method which handles this behavior. If that is the case, Graal can produce efficient JIT compiled code.

Most reactive languages which provide glitch-free propagation use a central manager which handles the propagation and uses a topologically ordered dependency graph and a priority queue [30]. These kinds of propagation algorithms are, however, not well-suited for the Reactive Ruby runtime system. This is the case since they are not well-suited for local optimization because in them behaviors do not directly communicate with each other. Instead, behaviors communicate with a central manager which controls the propagation for these behaviors. That increases the *distance* between behaviors. Besides that behaviors do not directly communicate, there is a discrepancy between closeness in the graph and execution order. If the graph contains a chain of behaviors, it is possible that these behaviors are not reevaluated directly after each user. Therefore, we decided not to use a propagation algorithm which uses a central manager.

Instead, the Reactive Ruby runtime system uses a modified version of the SID UP [15] algorithm, which was developed for reactive programming in a distributed setting. In this context, communication with a central coordinator is expensive. Therefore, behavior nodes are self-sufficient: they only communicate with their predecessors and successors in the dependency graph. This characteristic is well-suited for a speculative optimization with Truffle because optimization patterns only need to consider a behavior and its predecessor and successor behaviors. Therefore, an aggressive local optimization is possible.

4.2.1 The Algorithm

The propagation algorithm has two main functionalities. It needs to handle the insertion of new nodes and it handles the propagation of changes. The original SID UP algorithm can handle dynamic changes to the graph structure. In Reactive Ruby the dependency graph does not change dynamically, this is why this part of the algorithm is left out.

The propagation algorithm stores the following information in the behavior nodes.

Source Node
: A source node `s` stores an ID. The ID is a unique long value. It also stores all behaviors which depend on the source node `s`.

Behavior Node
: A behavior node `b` stores the ID of all source nodes s_i .. s_j which can reach `b`. In addition, behavior `b` saves the number of predecessors which each of the sources s_i .. s_j can reach.

The left part of Figure 7 visualizes a dependency graph with the stored information. Every source has a unique identification number. All behaviors save for each source that can reach them the number of predecessor nodes this source can reach. Source `1` and source `2` can reach behavior `a`. Source `1` can reach `a` directly and via `a`'s predecessor `b`, therefore `a` saves `ID:1 ; 2`. Source `2` can reach `a` via `a`'s predecessor `b`, therefore `a` additionally saves `ID:2 ; 1`. Source `1` and `2` both can reach `b` via exactly one predecessor, therefore `b` saves `ID:1 ; 1` and `ID:2 ; 1`. Only source `2` can reach `c` therefore `c` saves `ID:2 ; 1`.

Adding of New Nodes

When the runtime system creates a new behavior, it adds a new node to the dependency graph. The algorithm distinguishes between the insertion of a source node and a behavior node.

New source node
: The adding of a new source node is trivial. The system adds the new source node to the dependency graph. This node has no edges and it gets the next free ID.

  In the right part of figure 7 a new source is added (dashed triangle). The new source node gets the next free ID which is in this case 3.

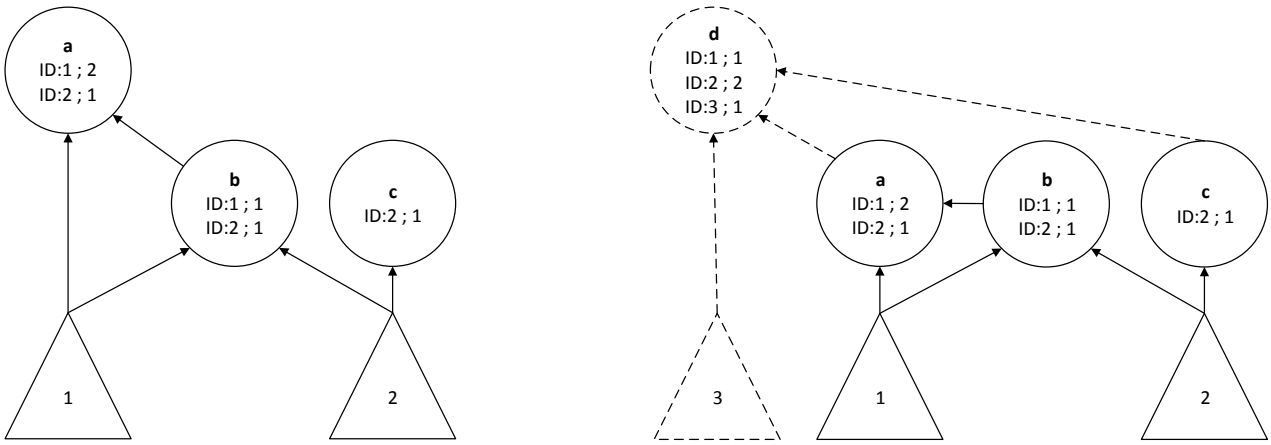

Figure 7: Dependency graph which visualizes the information stored for the propagation algorithm.

New behavior node

Adding of a new behavior node b_{new} is more complex. When the runtime system creates behavior b_{new}, it adds a new node to the dependency graph. All behaviors on which behavior b_{new} depends add an edge to the new behavior. After that, behavior b_{new} calculates from which source and via how many predecessors it is reachable. To do so, it iterates over all predecessors and collects all sources in a multi set. Then it counts the number of occurrences for each source in the multi set and saves these numbers.

In the right part of figure 7 a new behavior is added (dashed circle `d`). Behavior `d` depends on behaviors `3`, `a` and `c`. It collects from predecessor `3` the information that it is reachable from source `3`. Moreover, it collects from predecessor `a` the information that it is reachable from source `1` and `2`. Lastly, it collects from predecessor `c` the information that it is reachable from source `2`. This results in the multi set `{3,1,2,2}`. Therefore, `d` saves that source `1` reaches two predecessors (`ID:1 ; 2`), source `2` reaches one predecessor (`ID:2 ; 1`) and source `3` reaches one predecessor (`ID:3 ; 1`).

Propagation of Update Events

The propagation phase starts when the value of a source node changes. During a propagation phase no other source node is allowed to change. The propagation phases must happen mutually exclusive. In addition, the algorithm always processes exactly one node.

During a propagation phase, all nodes that are in the reachable closure of the changed source node are processed. Each node in the graph counts the number of change events it receives during one propagation phase. The `count` is 0 at the beginning of a propagation phase. Moreover, each node has a boolean flag (`changed`) which indicates if the node has changed. Initially, this flag is false. Figure 8 (change source node `1`) visualizes the data stored at the beginning of a propagation. In this figure all nodes have a `count` which is 0 and their `changed` flag is false. Furthermore, the nodes `a`, `b` and `d` are in the reachable closure of source `1`, which has changed.

The propagation phase starts when the changed source node starts to send out change events to all successor nodes. Source `1` sends a change event to `a` (change event 1 -> a, figure 8) and to `b` (change event 1-> b, figure 8). The change event that is propagated during a propagation phase contains the ID of the source node and the information whether the node that sends the change event was changed in the current propagation phase. In addition, it contains the behavior which changed most recently.

The remaining part describes the algorithm by explaining how a node handles an incoming change event:
During the propagation phase a node sets the `changed` flag to true when it receives a change event from a behavior whose value changed. In figure 8 (change event 1->a) node `a` revives a change event from node `1`, which changed its value this turn. Therefore, node `a` sets the `changed` flag to true. Moreover, when a node receives a change event, it increments its counter (`count`) by one. In Figure 8 (change event 1 -> a) node `a` receives a change event. Therefore, node `a` increases the `count` value to 1.

After a node has received a change event, the node checks for the source in the change event the number of predecessors via which this source can reach the node. If the number of received change events (`count`) is smaller than this number, the node waits for further change events. Source `1` can reach node `a` in figure 8 (change event 1 -> a) via two predecessors (`ID:1 ; 2`). The value of `count` for node `a` is 1, therefore it needs to wait for further change events. If the value of `count` is equal to the number of predecessors the source can reach, the node can execute its behavior expressions.

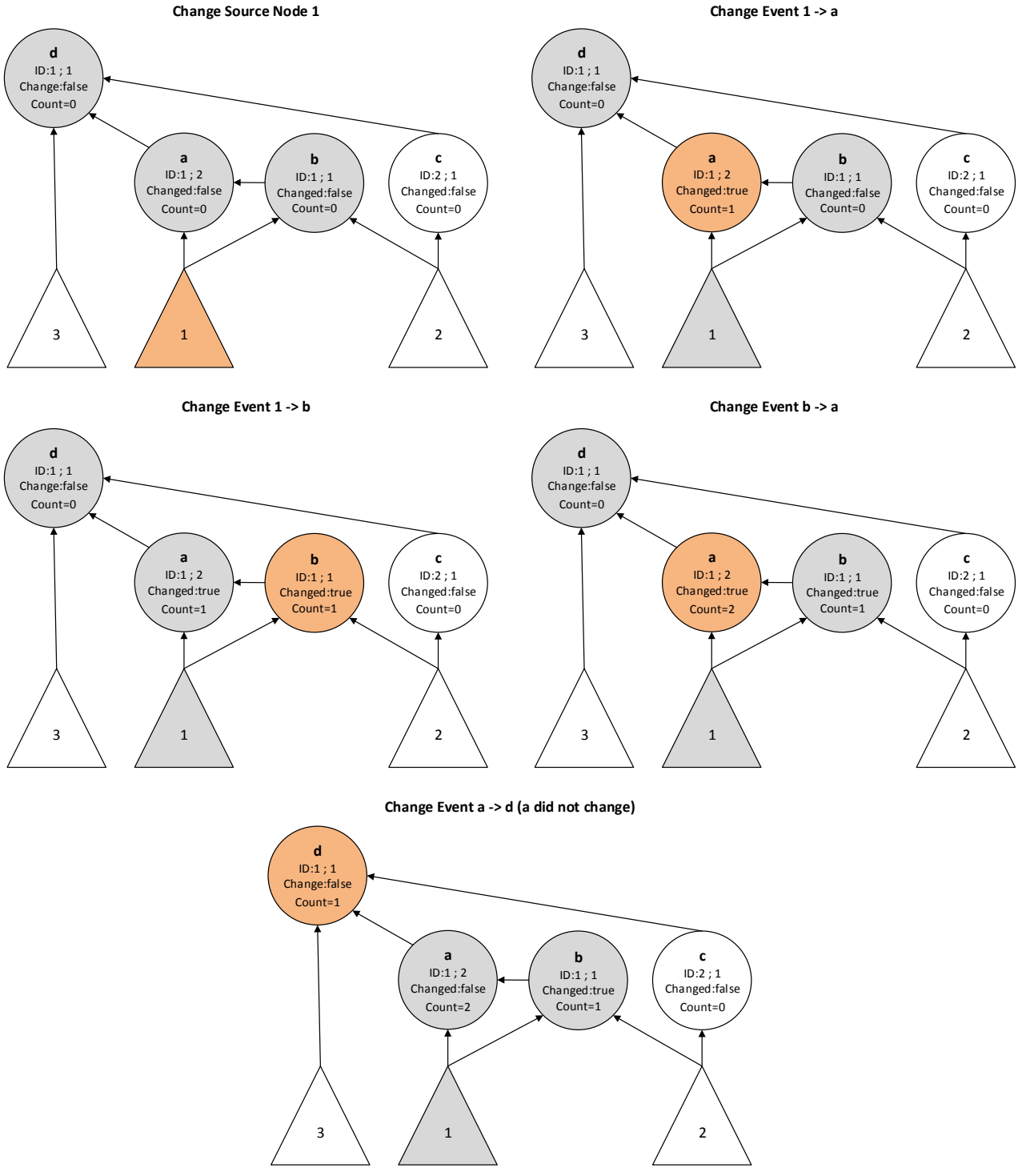

Figure 8: Dependency graph which shows the changes of stored information during the propagation of changes.

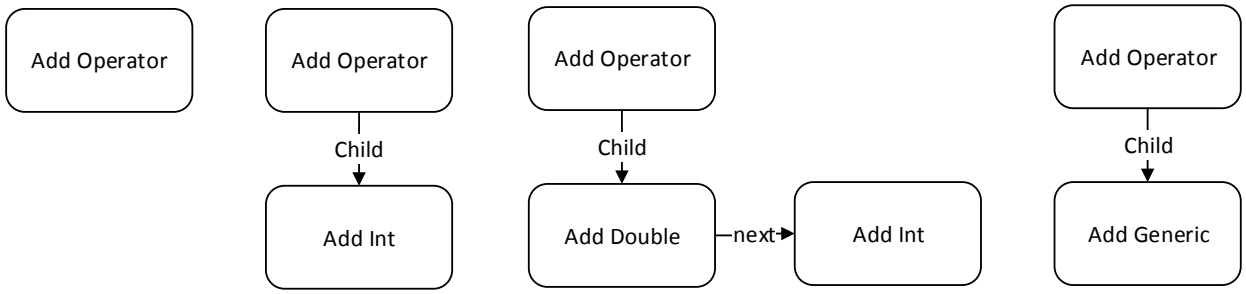

Figure 9: The evolution of a dispatch chain for the add operation. See listing 7 for the code of this example

Depending on whether the `changed` flag is set, the behavior expression needs to be evaluated and a new behavior value is saved. In figure 8 (change event 1 -> b) node `b` receives a change event and its `count` is equal to the number of relevant predecessors. In addition, the `changed` flag is true therefore it executes its behavior expression. Figure 8 (change event a-> d (a did not change)) illustrates the case of the `count` being equal to the number of predecessors the source can reach and the `changed` flag being false. Therefore, `d` does not need to execute its expression.

After a node has received all change events for one turn and maybe executed its expression the behavior sends a change event to all successor behaviors. The change event contains the information whether the behavior changed its value. In addition, it contains the source, which triggered the change propagation. Moreover, the change event contains the current behavior since it is the last behavior that changed. After that, the behavior sets the `count` to 0 and the `changed` flag to false. In figure 8 (change Event a-> d (a did not change)) node `a` notifies `d` after it has received a change event from `1` and `b`. The value of `a` did not change, therefore it sends the information that it has not changed to node `d`.

Correctness

The changed SID UP algorithm, as presented here, only works for reactive languages with static dependencies and it must be guaranteed that in a propagation turn exactly one source was changed. The algorithm could probably be modified in order to enable it to work with reactive languages which have dynamic dependencies.

As a consequence of the restrictions mentioned above, the check whether a node `b` receives `n` change events is equivalent to the check of the SID UP algorithm. Here, `n` is the number of predecessors of `b` which are reachable by the current source. That is because the SID UP algorithm checks every time it receives a change event for all predecessors which are reachable by the source node if they have changed. It only receives a change event if one of its predecessors which is reachable by the source was completely processed. A node changes exactly once. Therefore, a node is called `n` times during a propagation turn.

4.3 Dispatch Chains and Inline Caches

A dispatch chain [33] allows for chaining different speculative optimizations. Dispatch chains are a common pattern in self-optimizing AST interpreters [33]. The Truffle DSL [28] uses them to implement, among others, type specializations of operators. We use dispatch chains to implement optimizations for reactive programming.

Before this section explains the dispatch chain in detail, it provides a short example of the evaluation of a dispatch chain. The following example explains the evaluation of a dispatch chain from a fast speculative optimization to the slower non-speculative optimized state. Obviously, for a real application it is desired that the dispatch chain does not reach it generic state (end state).

```
1 + 3
4 + 3
3 + 3.4
"Hallo" + "Peter"
```

Listing 7: Program which performs addition on different types to demonstrate the evolution of the dispatch chain.

Figure 9 visualizes a dispatch chain for a type speculation of the addition operator for the example code in listing 7. Simplified, this dispatch chain handles all addition operators in the whole application. Before the interpreter executes the program, the dispatch chain is in an initial state. During the execution of line 1, the dispatch chain adds a node (`Add Int`) to handle the integer addition, which is a speculative optimization. The execution of line 2 is handled by this node (`Add Int`). In line 3 the operator adds an integer and a double number together which the `Add Int` node cannot handle.

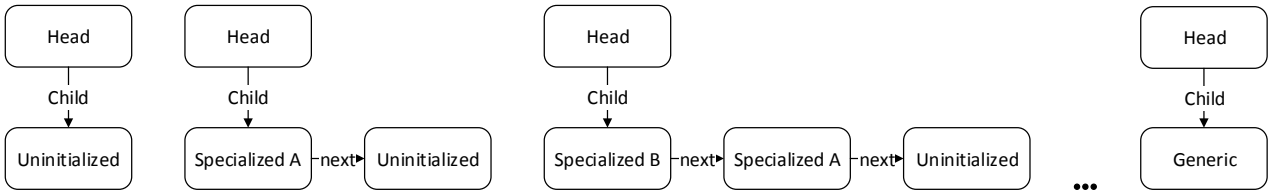

Figure 10: Evaluation of an arbitrary dispatch chain.

Therefore, Truffle adds a new node (`Add Double`) to the dispatch chain which is able to perform the addition of doubles, which is another speculative optimization. The dispatch chain is now able to perform the addition for the type int and for the type double. The code in line 4 adds two strings. Neither the `Add Int` node nor the `Add Double` node can handle the type string. The dispatch chain can add an `Add String` node which can handle the addition of strings. However, the dispatch chain reached a predefined max length, therefore Truffle replaces the chain with one node (`Add Generic`) which can handle the additions for all types. The `Add Generic` node replaced all speculative optimizations with a generic implementation of the addition operator.

A more general explanation of the dispatch chain follows now. Figure 10 displays an initial dispatch chain and its evolution. In general, a dispatch chain attempts to provide speculative optimizations for an operator. To allow for more than one speculative optimization of the operator, the dispatch chain allows for chaining nodes. Each node provides a speculative optimization and with each added node the dispatch chain provides speculative optimizations for more inputs. However, with each node added the code size increases and the performance of some optimizations (nodes in the back) decreases. Therefore, at some point when the dispatch chain gets too long, the whole dispatch chain is replaced by one node which implements the semantics of the operators.

Now an explanation which focuses on the implementation follows. A dispatch chain contains a `head` node, which hides changes inside the dispatch chain from the outside. When the system executes the `head` node, it calls its current child node to handle the input. Initially, the child node is the `Uninitialized` node, which creates new nodes and inserts them in the dispatch chain. The `Uninitialized` node performs all node rewrites which happen in the dispatch chain. Truffle never compiles the `Uninitialized` node because it performs node rewrites, which are only allowed in the interpreter.

The first n times the `Uninitialized` node is executed, it creates specialized nodes which provide a speculative optimization for the input that reaches the `Uninitialized` node. In figure 10, the first time the system executes the `Uninitialized` node it creates the `Specialized A` node and the second time it creates the `Specialized B` node. The `Uninitialized` node usually inserts the new specialized node as a child of the head node. In figure 10 the third chain the `Specialized B` node is inserted before the `Specialized A` node.

The specialized nodes implement speculative optimizations and cannot handle all inputs. Therefore, they contain checks which ensure that they can handle the provided input. In addition, specialized nodes can contain assumptions, with whom the system can invalid a specialized node. When a specialized node is able to handle the current input, it executes its implemented semantics. If it cannot handle the input, it redirects the execution to the next node (executes the next node) in the dispatch chain.

When the dispatch chain gets too long (longer than a predefined number), the `Uninitialized` node replaces the whole chain with a `Generic` node (the last dispatch chain in figure 10). This node is able to handle the whole semantics of the operator. This `Generic` node is slower than the specialized implementations and it ideally is created as rarely as possible.

In general Truffle can trigger JIT compilation for each state of a dispatch chain. However, the specialized nodes are usually significantly faster than the generic nodes. The concept that Truffle clones a method and copies it to a call site before it inlines the methods (as explained in 3.2.2) helps to reduce the number of generic nodes in dispatch chains.

4.3.1 Inline Chaches

The Reactive Ruby runtime system uses the Polymorphic inline caches (PIC) [27], which Truffle Ruby provides. A PIC is a technique to speed up method calls. For that it, saves previous called methods at the call site and contains direct call to these methods. PIC can be implemented in the form of a dispatch chain and they play an important role to provide an efficient runtime system. This is why they will be described in this section.

The Truffle Ruby runtime system distinguishes between *normal* method calls and Ruby block/proc/lambda calls. Reactive Ruby uses Ruby methods to implement the propagation logic. Moreover, it uses Ruby blocks for the implementation of the behavior expression. Therefore, the PIC for method calls and the PIC for block calls are important in the runtime system.

As PIC can be implemented in a dispatch chain, we explain the PIC by describing the specialized and generic nodes. In addition, we describe how these nodes work and we explain the uninitialized node.

The dispatch chain which implements a PIC for a Ruby method has one specialized node. This node caches a method and contains a direct call to that method. The generic node does a method lookup and then performs a normal call to that method.

To be more precise, the Ruby PIC has some additional specialized nodes for corner cases. E.g., one corner case is a method call to a non-Ruby method. These special cases are important for Ruby, but not for the reactive part. Therefore, we will skip them here.

Cached Dispatch Node

The cached dispatch node (specialized node) stores a method and has `DirectCallNode` node for this method as a child. The `DirectCallNode` is a Truffle API node and the Graal JIT compiler has special knowledge about this node. For this node, Graal can perform inlining as well as call site sensible cloning of the call tree [1, 28].

In addition to the method, the node stores an assumption which invalidates the code when the method is changed. It furthermore saves the class in which the method is implemented.

When the system executes the cached dispatch node for a method `m`, the node checks if the stored method is equal to the method `m`. If this is the case, the cached dispatch node performs a direct call for the cached method. If it is not the case, it redirects the execution to the next node in the dispatch chain.

Uncached Dispatch Node

This node is the generic node in the dispatch chain. It performs a method look up whenever it is called. For that, it checks if the class of the object on which the method is called defines that method. If this is the case it loads that method and calls this method. If this is not the case, it traverses all ancestors of the class until it finds the method. The method look up requires runtime information and will not be JIT compiled. As a result, the uncached dispatch node is slow compared to the cached dispatch node.

Unresolved Dispatch Node

The unresolved dispatch node (uninitialized node) creates cached dispatch nodes as long as the chain is shorter than the max PIC chain length. To create a cached dispatch node, it performs a method look up as explained in the paragraph above and creates the new node for this method. If the chain is too long, it replaces the dispatch chain with one uncached dispatch node.

PIC for Ruby Blocks

Overall, the PIC for Ruby blocks work in the same way as the PIC for methods. The main implementation difference is, that the generic node does not need to perform a method look up. The semantics of block and method calls differs, therefore they both need different implementations. However, the implementation of the PIC for Ruby blocks is similar to the PIC for methods. Therefore, we do not consider a further explanation to be beneficial here.

4.3.2 Use of Inline Caches in Reactive Ruby

Reactive Ruby neither performs explicit JIT compilation of the propagation logic nor explicitly merges chains of behaviors into a single behavior. The implementation of Reactive Ruby uses PIC in the propagation algorithm. This use of PIC together with node rewrites creates efficient code for the propagation logic. Furthermore, these both together enable Truffle to generate compiled code for, e.g., chains of behaviors which does not contain calls and almost no propagation logic.

It is not obvious how the implementation of the propagation of changes (section 4.6) enables merging of behaviors during the propagation of changes. Therefore, we explain here how the PIC which are used in the propagation of changes, enable Truffle to perform this operation.

Reactive Ruby uses a propagation algorithm in which every node locally performs the functionality to receive change events, ensure glitch-freedom, reevaluate the behavior expression and notify successor nodes. Every behavior shares a method which handles all the functionalities listed above. Figure 11 visualizes this method, for explanatory reasons we name this method *propagation method*.

In Truffle, all methods are implemented as AST trees. Therefore, we use *subtrees* to describe a part of the AST of a method which performs a certain task. The tree of the `propagation method` contains a subtree which processes the incoming change events and checks if the behavior needs to wait for further change events to guarantee glitch freedom (`Glitch Freedom Logic`). The second subtree (`Reevaluate Behavior Expression`) executes the behavior expression, changes the behavior value and notifies handlers if necessary. The last subtree (`Propagation Changes`) sends out change events to the successor nodes in the dependency graph.

We use PIC in the `reevaluate behavior expression` subtree and in the `propagation changes` subtree. The implementation uses inline caches for the execution of Ruby blocks, which are used by some behavior expressions (`Reevaluate`

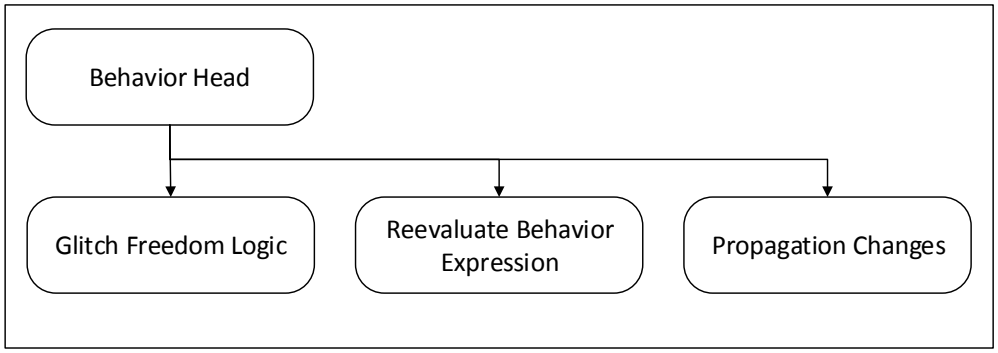

Figure 11: The AST of the `propagation method` which handles the propagation of changes in Reactive Ruby. The nodes glitch freedom logic, reevaluation of the behavior expression and propagation changes represent subtrees of these AST.

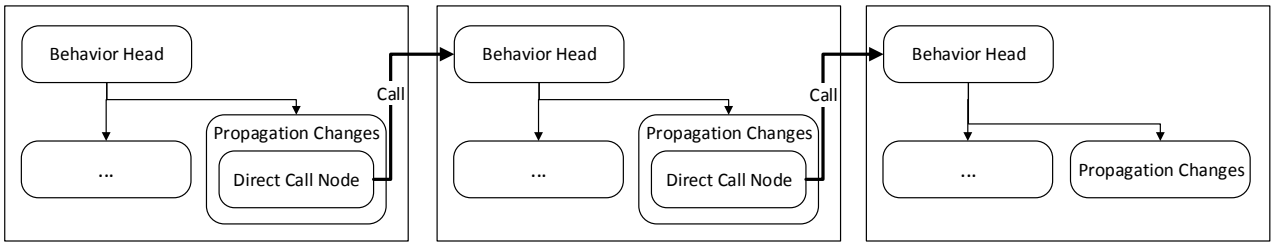

Figure 12: Chain of 3 behavior nodes with specialized PIC in the propagation code.

`Behavior Expression` subtree). In addition, it uses PIC to call the successor behaviors (`Propagation Changes` subtree), which allows combining chains of behaviors.

Now we explain how PIC enables merging behaviors during propagation. Therefore, the `propagation changes` subtree is interesting. Figure 12 visualizes a chain of three behaviors. In this chain the first two behaviors contain a PIC which already has the specialized node `DirectCallNode`. This node caches the `propagation method` of the next behavior and has a direct call to that method. This direct call allows Reactive Ruby to inline this method of the next behavior as well as to clone it.

Inlining removes the call between behavior nodes during propagation. In other words it merges two behaviors during propagation. However, the cloning of the `propagation method` is also very important for the following reason. The `propagation method` provides an efficient optimization for, among others a behavior which is part of a chain structure. However, initially all behaviors share the `propagation method`. Therefore, this method will likely be in a generic state. When the method is cloned for the first behavior node in the chain structure, Truffle sets the cloned method in its initial state and it can specialize for that call site (specialize for a chain structure). In figure 12 the cloned `propagation methods` can specialize to a version which works well for a chain structure. In conclusion, the cloning of the `propagation method` together with the optimization of these methods provides the speculative optimization of the code.

After Truffle creates the direct calls, cloned the methods and optimized the cloned methods, it can inline the calls, and then Graal can JIT compile it. After inlining, the chain of three behaviors looks simplified as in figure 13. Truffle provides these inlined methods in an intermediate representation to Graal. Then Graal can JIT compile it to efficient code.

4.4 Behavior Object

An important corner stone of Reactive Ruby is the `behavior object`, which is the runtime object that represents a behavior. Every reactive operator works with it. Since almost the complete Reactive Ruby code depends on the `behavior object`, it will be explained in more detail than other parts of Reactive Ruby. We also hope that explaining the `behavior object` more thoroughly will help a reader who is unfamiliar with Truffle to better understand some ideas behind it.

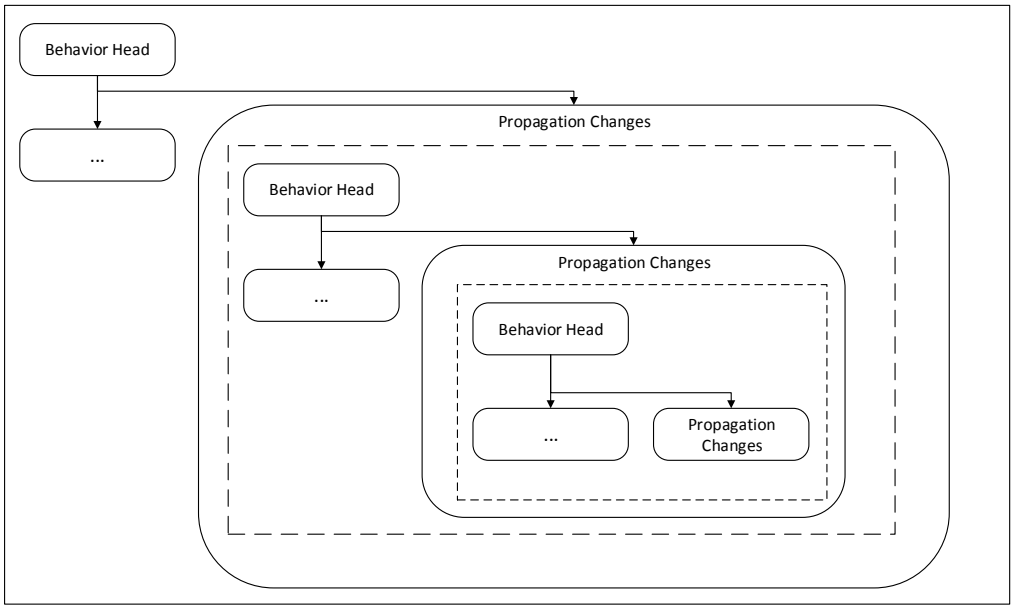

Figure 13: Conceptional overview of an inlined chain of 3 behaviors.

The `behavior object`, like most non-primitive values in Truffle Ruby, is implemented in a Java class. Truffle allows arbitrary Java classes to describe a value in the language. To represent these values in the interpreter, Truffle creates objects from these classes. The following code is a naive but valid implementation of the data saved in a behavior.

```
public class Behavior{
        Object value;
        Object behaviorExpr;
}
```

This implementation is slow because it saves the value of a behavior in an object. A behavior can hold an arbitrary value, for example a string or an int. When the system saves a primitive value in an object, it needs to box this value before storing. To box a value before storing and to unbox a value before reading adds an overhead to the access of the value. Therefore, Reactive Ruby provides specializations for behaviors which contain primitive values. To allow this specialization, the `behavior object` must be able to store primitive values without boxing. Therefore, the `Behavior` class can contain a two int attribute and an object attribute for the value.

```
public class Behavior{
        int valueP1;
        int valueP2;
        Object valueO;
        ...
}
```

In this implementation, the interpreter (or the compiled code) can store primitive values in `valueP1` and non-primitive values in `valueO`. In this implementation, the attribute `valueP1` would not only represent int values but also, e.g., boolean values. Unfortunately, this is not enough. A behavior can hold, for example, a long value. Therefore, the `Behavior` class contains a second int attribute `valueP2` and a long value is saved in `valueP1` and `valueP2`. An efficient implementation of the reading and writing functionality for object attributes is complex. Therefore, the `behavior object` uses the Truffle object storage model (OSM) [43] which provides the functionality to efficiently read and write to object attributes.

Reactive Ruby uses the OSM to save, for example the value a behavior holds. A class that implements a runtime value can use the OSM and attributes together to describe this value. A still simplified version of the `behavior object` is the following. This version of the `behavior object` uses the OSM to store the value held by a behavior and the attribute `successors` to save the successor behaviors.

```
public class Behavior{
        private final DynamicObject dynamicObject;
```

```
        Behavior[] successors;
        ...
}
```

The attribute DynamicObject provides an interface to Truffle OSM. Reactive Ruby uses the OSM only for some information. In more detail, Reactive Ruby saves information which the language uses and information which are not shared by all behaviors in the OSM. For example, an application can read the value hold by a behavior, therefore it is saved in the OSM. Information which are only used in the runtime system are saved as normal attributes in the Behavior class because that reduces the implementation complexity and does not create a significant performance difference.

Listing 8 shows the important parts of the description of the `behavior object`. The explanation of the operators (section 4.5) and the description of the change propagation implementation (section 4.6) use the `behavior object`, therefore it is explained in detail here.

- Line 3 adds the dynamicObject (interface to the Truffle OSM) to the `behavior object`. Reactive Ruby saves the value a behavior holds in the OSM. In addition, the `behavior object` uses it to stores information which only some special behaviors use, e.g., the take operator saves a counter variable in it.
- The attribute successorBehaviorNode (line 4) stores all successor nodes in the dependency graph.
- The attributes functionStore (line 6) and functionStoreSize (line 7) store the handlers, which are Ruby blocks. The system calls these blocks when the value a behavior holds changes.
- The attribute id (line 9) is the ID of a behavior. All behaviors, and not only the source behaviors, have an ID. The ID is useful for debugging and error handling.
- The attribute sourcePreCount (line 11) stores the ID of all sources that can reach this behavior. Likewise, the attribute sourcePreCount saves for each source the number of predecessors this source can reach. The annotation @CompilerDirectives.CompilationFinal provides Graal with the information that it can consider the annotated attribute as constant in the compiled code.
- The attribute chain (line 12) is a flag which indicates that this behavior only depends on one behavior. The runtime system uses this flag to create specialized propagation code.
- The attribute type (line 14) is a flag that provides information about which behavior expressions a behavior has. For example, it can indicate that the behavior expression is the `map` operator. This flag is important for the optimization of certain behavior expressions.
- The attributes count (line 16) and changed (line 17) are used during the propagation phase. For more information, one can read about propagation algorithm in section 4.2.1.

```
1  public final class BehaviorObject extends RubyBasicObject {
2          //inherited from RubyBasicObject
3          DynamicObject dynamicObject;
4          private BehaviorObject[] successorBehaviorNode;
5
6          private Object functionStore;
7          private int functionStoreSize = 0;
8
9          private final long id;
10
11         @CompilerDirectives.CompilationFinal long[][] sourcePreCount;
12         @CompilerDirectives.CompilationFinal boolean chain;
13
14         @CompilerDirectives.CompilationFinal int type;
15
16         private int count = 0;
17         private boolean changed = false
18
19 }
```

Listing 8: The Behavior Class.

The BehaviorObject class is used to create both the runtime value of a source behavior and a non-source behavior.

4.5 Operators

The implementation of the operators consists of two components. The first component is a method which creates a new behavior (`behavior object`) whose behavior expression represents this operator. The second component implements the functionality of the operator. It is executed during the change propagation algorithm.

Creation of New Behaviors via Operators

Methods of the behavior class and the source class implement the component of an operator which creates new behaviors. Most operators work in the same way for behaviors and sources. Therefore, both classes share the method implementation if possible.

Operators create new behaviors (`behavior object`) and can add attributes to the behavior via the OSM. Subsequently, they initialize the added attributes and add the new behavior to the dependency graph. To do so, the operators add the new behavior to the list of successors of all behaviors on which it depends and it adds all these behaviors as its predecessors. Then, an operator calculates the set of sources which can reach the new behavior and calculates the number of predecessors each of these sources can reach.

After the operator has added the new behavior to the dependency graph and has added the necessary attributes, it calculates the initial value of the new behavior.

The Functionality of the Behavior Operators

The operators are implemented in small trees. This is contrary to classical operators like, for example, `+` which are implemented in single nodes. These trees implement the functionality which is needed to reevaluate the behavior expression created by the operators and to save the new value.

In general, a behavior operator needs to read the values of the predecessor behaviors. After that, it recalculates the behavior expression and updates the value of the current behavior.

Some behavior operators depend on exactly one predecessor. This is for example the case with the `filter` operator. In this case, the operator can access the value of the predecessor behavior from the change event. The other case, in which a behavior operator depends on more than one predecessor, is slightly more complex. First, the operator needs to access the list of predecessors. Then, it must read the values of all predecessors. After that, it can reevaluate the behavior expression and save the new value.

4.5.1 Map Operator

This chapter only describes the `map` operators in more detail because the implementations of the other operators are similar.

We chose to explain the `map` operator because it has a clear and simple semantics. The `map` operator of a behavior b takes as parameters a number of behaviors $b_1, \ldots, b_n$ and a Ruby block `f`. It creates a new behavior b_{new} which depends on the behavior b and the behaviors $b_1, \ldots, b_n$. The value of behavior b_{new} is the Ruby block `f` applied to the values of b and $b_1, \ldots, b_n$.

The `map` operator can create two different behaviors. The `map` operator of b can either be called with the parameters $b_1, \ldots, b_n$ or without the parameters $b_1, \ldots, b_n$. For reasons of convenience, we name the `map` operator without the parameters $b_1, \ldots, b_n$ `map0` and the `map` operator with the parameters $b_1, \ldots, b_n$ `mapN`. If the explanation for `map0` and `mapN` is the same, we use the name `map`.

Operators in Reactive Ruby are implemented in two components. One component handles the creation of new behaviors (`behavior object`) and its initialization. The second component handles the recalculation of the operator in the propagation phase.

Creation and Initialization of a New Map Behavior

The creation and initialization of a new behavior (`behavior object`) is currently an interpreter only operation. This means that Graal will not JIT compile this code. The following explanation will use the object layout of the `behavior object` (listing 12). The `map` operator performs the following steps:

- It creates a new `behavior object`. The flag `type` is set to `map0` or `mapN`.
- It inserts the new `behavior object` in the dependency graph. For that, the `map` operator adds the new behavior to the list of successor nodes of behavior b. The `mapN` operator also inserts the new behavior in the list of successors nodes of the behaviors $b_1, \ldots, b_n$. Furthermore, the `mapN` operator saves the nodes b and $b_1, \ldots, b_n$ as its predecessor nodes. The predecessor nodes are saved in the `dynamicObject` of the new behavior.
- It calculates the sources which can reach the new behavior. For each source it calculates the number of predecessor it can reach. This information is stored in `sourcePreCount`. The `map0` operator sets the `chain` flag to true.

- The operator saves the Ruby block `f` in the OSM (`dynamicObject`). This is an operator specific attribute. Not all operators need to save a Ruby block.
- The `map0` operator reads the value of behavior `b`. The `mapN` operator reads the value of behavior `b` and behaviors $b_1, \ldots, b_n$.
- The operator executes block `f` with the value of the behaviors as its arguments, which it read in the last step.
- It saves the return value of block `f` in `dynamicObject` as the value it holds.
- Lastly, it returns the new behavior.

Recalculation of the Map Operator during Change Propagation

When the runtime system recalculates a behavior expression of a behavior which is created by the `map` operator during the change propagation phase, the following steps happen.

- The `map` node reads block `f` from the `dynamicObject` (OSM).
- If the operator is `map0`, it reads the value of the behavior which is provided inside the change event. If the operator is `mapN`, it accesses the predecessor behaviors from `dynamicObject`. Subsequently, it reads the value of each of the predecessor behaviors.
- It calls block `f`. It uses the values which it read in the last step as arguments for the block call.
- It saves the result of the block call as the new value the behavior holds.
- If the new value is different from its old value, it notifies the change propagation logic that it has changed.

4.6 Implementation of the Change Propagation

The implementation aims to fully utilize inline caches to provide an efficient runtime system. Therefore, every behavior node in the dependency graph locally handles the functionalities necessary for the propagation of changes. Furthermore, the propagation of changes works without a central manager. The entry point to change propagation logic is the `propagation method`. All behaviors use this method to handle the propagation. Figure 11 visualizes the AST of this method. As an exception, source nodes have a different method which handles the propagation. However, it is similar and will therefore not be explained in detail.

The `propagation method` handles the whole change propagation for a behavior (`behvior object`). This method ensures glitch freedom, executes the behavior expression and handles the notification of registered handlers. In addition, it notifies the successor behaviors of the behavior.

The behaviors communicate during the change propagation by calling the `propagation method` on successor behaviors. When a source changes, it calls this method on all successor behaviors. If a behavior changes, it likewise calls this method on all successor behaviors.

In the propagation phase, behaviors pass information (change event) to their successor behaviors. Behaviors pass this change event via the arguments of the `propagation method` call. The information which is passed to the successor nodes during change propagation is the following:

- **Source ID:** The source ID stores which source node was changed in the current propagation turn.
- **Last Node:** The behavior which sends the change event.
- **Changed:** The information whether the behavior which sends the change event changed its value.

The implementation of the `propagation method` is organized in three main components. Each of these three components is a subtree of the AST which implements the `propagation method`. One component checks when the behavior receives a change event if it needs to wait. A behavior can receive multiple change events in one propagation turn and it must wait until it receives the last change event. This component purpose is to ensure glitch freedom.

Another component (subtree) reevaluates the behavior expression and it handles the order of change events in case they happen simultaneously. In addition, it saves the new value of the behavior and it notifies handlers if handlers are registered.

The last component (subtree) sends out a new change event to all successors of the current behavior. For that, it first creates a new change event which contains the source that started the propagation phase. In addition, it contains the

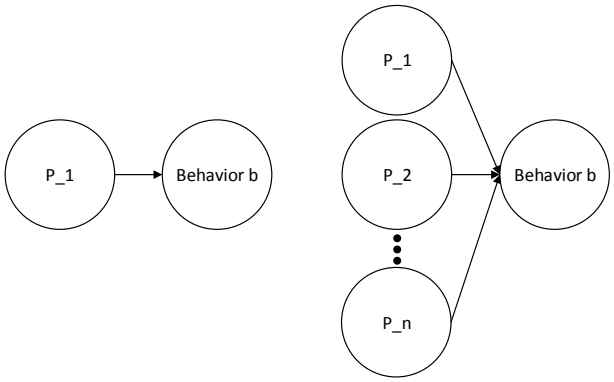

Figure 14: The left figure shows a part of a dependency graph in which a behavior only depends on one predecessor, and the right figure shows a part of a dependency graph in which a behavior depends on an arbitrary number of predecessors.

current behavior as the last node in the change event. Moreover, it comprises the information whether the behavior value of the current behavior was changed.

Initially, behaviors share the `propagation method`. However, tree cloning allows the duplication of this method for some behaviors. Section 4.3.2 explains this process in more detail. Here is important that some behaviors share the `propagation method` and some behaviors have their own version.

The rest of this section explains the subtrees of the `propagation method`.

4.6.1 Glitch Freedom Logic

This subtree of the `propagation method` ensures glitch freedom. It uses a dispatch chain (section 4.3) to provide speculative optimizations for certain graph structures. In the propagation algorithm a behavior only depends on its predecessor and successor behaviors. To guarantee glitch freedom only the predecessor nodes are important, all predecessor nodes of the behavior in the dependency graph need to be in their final state for the current propagation turn. This is the case if they have already been reevaluated or if they are in the set of nodes which do not change in the current turn.

There are two local graph structures which are distinguished here (figure 14). The behavior can have exactly one predecessor (figure 14, left graph). Alternatively, the behavior can have an arbitrary number of predecessor nodes (figure 14, right graph).

Reactive Ruby provides one specialized dispatch chain node for the chain structure (figure 14, left graph). For the non-chain structure (figure 14, right graph) it provides a specialized dispatch chain node and a generic dispatch chain node.

Specialized Chain Node

For a behavior which has one predecessor behavior the glitch freedom check is trivial. Since the behavior has one predecessor behavior, it receives one change event at most in every propagation turn. The only task of the specialized chain node for a chain structure is therefore to check if the current behavior has one predecessor behavior. Every behavior has a flag which indicates if the behavior has exactly one predecessor. The specialized chain node checks this flag and if the flag is true, the execution of the subtree `execute reactive code` can start. If the current behavior has more than one predecessor, the specialized chain node executes to the next node in the dispatch chain.

Specialized Non-Chain Node

The specialized non-chain node provides a fast glitch-free check for a cached position. It performs the glitch-free check for each change event the behavior receives, in O(1) with a small constant overhead. Therefore, it cannot perform the same check as performed in the SID-UP algorithm because that check needs to iterate over all predecessor nodes and performs a calculation for each predecessor.

Every behavior saves the ID of each source which can reach that behavior in an array. In that array, the behavior also saves the number of predecessors via which each source can reach this behavior. The specialized non-chain node stores the position where in the `behavior object` a source ID and the number of predecessors this source can reach is stored.

When the specialized node is executed, it checks the following:

(1) The specialized non-chain node checks if the saved position is valid for the current behavior.

(2) The specialized non-chain node checks if the source ID of the current change event is equal to the source ID in the cached position.

(3) The specialized non-chain node checks if the number of the predecessor paths is equal to the number of change events the behavior received in this propagation phase.

If all three conditions are true, the behavior will not receive any more change events. Glitch freedom is achieved for this behavior and the system can execute the subtree `reevaluate behavior expression`. If only conditions (1) and (2) are true, the node increments the counter of received change events. In this case it stops the execution and waits for the next change event. If condition (1) or condition (2) is not valid, this specialized dispatch chain node is not able to handle the current change event. Therefore, it executes the next node in the dispatch chain.

Generic Node

When the generic node receives a change event from the source `s` it does the following: It first searches for the position where the behavior saves the number of predecessors reachable by the source `s`. Then, it checks if the number is equal to the number of received change events in the current propagation phase. If that is the case, the behavior will not receive any more change events and the subtree `reevaluate behavior expression` can be executed. If the behavior still needs to wait for further change events, it will increment the counter. After that, it stops the execution and waits for the next change event.

Unresolved Node

The unresolved node creates the specialized dispatch chain nodes above. When the unresolved node is executed, it first checks how long the dispatch chain is. If the chain is longer than a predefined threshold, it creates a new generic node and replaces the whole dispatch chain with the new node. If there is still space in the dispatch chain, it creates one of the specialized nodes. If the current behavior has one predecessor, it creates the specialized chain node. If the current behavior has more than one predecessor, it creates the specialized non-chain node. To do so, the node searches for the position where the current behavior saves the source ID of the current change event. After that, it creates a specialized non-chain node for that position. When the unresolved node has performed its node replacement, it redirects the execution to the newly created dispatch chain node.

4.6.2 Reevaluation of the Behavior Expression

This subtree manages the execution of the behavior expression and the saving of the new behavior value. A dispatch chain implements the functionality. There is a specialized node which caches the different reactive operators and one generic node which can handle all operators. The different operator nodes read and write different attributes from the `behavior object`. Section 4.5 provides further information about how the different operators are implemented in Reactive Ruby.

The cached behavior operator is the specialized node in the dispatch chain. The uncached behavior operator node is the generic node in the dispatch chain.

Cached Behavior Operator

The cached behavior operator node caches a behavior operator, for example the `map` operator. When this node is executed for a behavior, it checks if it cached the right operator. In order to do so, it checks if the type of the behavior is equal to the type of the cached operator. If it has cached the right operator, it executes the cached functionality. If it did not cache the right operator, it executes the next node in the dispatch chain.

Uncached Behavior Operator

The uncached behavior operator node is the generic node in the dispatch chain. When this node is executed for a behavior, it checks which operator this behavior has. It then calls the operator which can handle the current behavior.

Uninitialized Node

The uninitialized node creates the specialized node cached behavior operator and the generic node. When it is executed, it first checks how long the dispatch chain is. If the dispatch chain is longer than a predefined threshold, it creates the generic node and replaces the whole dispatch chain with the generic node. If there is still space in the dispatch chain, the uninitialized node creates the cached behavior operator node. To do so, it checks which operator the behavior has, looks up that operator and then creates the specialized node for this operator.

After the uninitialized node has performed its node replacement, it redirects the execution to the newly created node.

Notify Handlers

This subtree calls Ruby blocks after the value of a behavior has changed and is part of the `revalue behavior expression` subtree (figure 11). It is specialized on how many handlers a behavior has. It provides a specialization for zero and for one handler. Additionally, a generic node which can handle an arbitrary amount of handlers exists.

Zero Handler: In case a behavior has zero handlers registered, the specialized dispatch chain node is simple, it does not need to call any handler. This node checks for a behavior if this behavior has no registered handlers, in this case the notification process is finished. Otherwise, the zero handler node executes the next node in the dispatch chain.

One Handler: The specialized node for one handler needs to call this handler. When this node is executed for a behavior, it checks if the behavior has one registered handler. If that is the case, it executes the handler. Otherwise, this node cannot handle the current behaviors and it executes the next node in the dispatch chain.

Arbitrary Number of Handlers: The generic node can call an arbitrary number of handlers. When the node is executed, it reads the handlers from the behavior. Then it iterates over the registered blocks and calls them.

Uninitialized Node: This node creates the specialized nodes. First, it checks if the dispatch chain is too long. If this is the case, it creates the arbitrary number of handler node. Otherwise, it checks how many handlers are registered at the current behavior. Depending on the number of register handlers, it creates the zero handler node, one handler node or an arbitrary number of handlers node and inserts the new node in the dispatch chain. After that, the uninitialized node executes the newly created node.

4.6.3 Change Event Propagation

The last subtree of the `propagation method` handles the notification of dependent behaviors. The subtree provides specializations for different numbers of successor behaviors. To do so, it uses a dispatch chain.

The specialized nodes as well as the generic nodes use a PIC to call the `propagation method` on successor behaviors. Especially in case of one successor behavior this PIC allows inlining.

Constant Number of Successors

This specialized node creates an optimization for a constant number of successor behaviors. The node is specialized for N successor behaviors and uses a PIC to call the successor behaviors.

When the system executes the node, it checks if the number of successor behaviors for the current behavior is equal to N. If this is the case, it calls the successor behaviors. For that, it has an unrolled loop in which it reads the successor behaviors and then calls the `propagation method` for each of these behaviors. This also creates an efficient code for a behavior which only has one successor. If the number of successor behaviors of the current behavior is not equal to N, this node execution the next node in the dispatch chain.

Variable Number of Successors

The generic node can notify an arbitrary number of successor behaviors. When it is executed for a behavior, it iterates over the successor behaviors. It calls the `propagation method` for each successor behavior.

Uninitialized Node

The uninitialized node creates the specialized nodes. First, it checks if the dispatch chain exceeds the predefined max length. In this case it creates the variable number of successors node. If that is not the case, it checks how many successor behaviors the current behavior has and then creates the constant number of successors node with N being the number of successor behaviors. After that, the uninitialized node executes to the newly created node.

4.7 Design Choices and Limitations of Reactive Ruby

To conclude the presentation of Reactive Ruby, we briefly discuss how its features fit into the landscape of reactive programming languages presented in Chapter 2.

Glitch freedom
: Reactive Ruby is a glitch-free reactive language. The propagation algorithm of Reactive Ruby guarantees a glitch-free propagation and does not evaluate unnecessary behavior expressions in a propagation turn.

Lifting
: Reactive Ruby uses a mixed strategy when it comes to lifting. In behavior expressions, the developer needs to perform manual lifting. He needs to access the value of a behavior via the operator `value`. Then he can work with the value inside the behavior expression. In addition, Reactive Ruby offers explicit lifting. The developer can use a number of operators to lift a Ruby block so that it can work with behaviors. Reactive Ruby, for examples offers the `map` operator which allows lifting a block which then can work with the value of one or more behaviors.

Unfortunately Reactive Ruby does not provide any Ruby operators which it already lifted. The implementation of a lifted version of, for example + is not difficult. However, time consuming.

Evaluation model
: Reactive Ruby uses a push-based evaluation model which applies eager evaluation. Whenever a behavior receives a change event and glitch-freedom is achieved, it will evaluate its behavior expression and update its value.

Dependency graph
: The dependencies in Reactive Ruby are static. The semantics of Reactive Ruby only allows behaviors which predecessors do not change.

4.7.1 Limitations

The current implementation of Reactive Ruby has a number of limitations that we plan to relax in future work. In the final sections, we provide details about the simplifications we made in Reactive Ruby. An outlook on future research is presented in section 6.2.

Cyclic dependencies
: In Reactive Ruby, cyclic dependencies are not allowed. The syntax permits the creation of cyclic dependencies, but they result in a runtime error. Therefore, the developer need to ensure that he does not create cyclic dependencies.

Single sources of changes
: A propagation phase in Reactive Ruby can only handle one source change at a time. Once a propagation turn starts, no source is allowed to change. In a multi-threaded environment a global lock needs to be used to guarantee that. Therefore, Reactive Ruby is complex to use in a multi threaded environment.

High-order reactive and behavior expressions with dynamic dependency discovery
: Reactive Ruby does not support higher order behaviors. Furthermore, it does not perform dynamic dependency discovery in the behavior expressions. As a consequence, the dependency graph is static. With that, the expressibility of Reactive Ruby is lower than, for example Scala.React [30].

Limitation of the discovery of dependencies in behavior expression
: There is a limitation to the identification of dependencies in the behavior expression. Reactive Ruby only discovers behaviors inside the `behaviorexpr` which are directly read from a local variable. Behaviors which, for example, are read in method calls or returned by methods are not identified.

  In more details, when the system creates a behavior expression, it iterates over all local variables which are read in the block and checks if they are defined in the closure of the block. For all variables that are defined in the closure of the block, the system checks if the variable holds a behavior. All behaviors identified this way are the system identifies as predecessors of the new behavior.

  In conclusion, Reactive Ruby does not discover e.g., behaviors which are read from instance variables.

Limitation in the optimization
: Reactive Ruby starts the optimization in the change propagation at points where the reactive code interacts with the imperative code. That means that Reactive Ruby starts these optimizations at a call site to the `emit` method. A consequence is that different sources do not share optimizations. In other words, every source node starts its own optimization at the call site of the `emit` method and only this call site uses these optimizations.

  The reason for this limitation is that all behaviors initially share the `propagation method`. Therefore, this method is likely to be in a generic state. Generic state here means that most dispatch chains in this method do not provide specializations and use the generic nodes to perform the semantics. Truffle can JIT compile this method when it is in the generic state, however, it does not contain the fast speculative optimizations. Truffle needs to clone the `propagation method` so that the clone can provide speculative optimizations. The point where Truffle clones methods is in the `DirectCallNode`, which is used in cached dispatch node which is a specialized node in the PIC. The `emit` method is small and Truffle can inline and clone this method for a call site. This cloned method then has a high chance to contain a cached dispatch node for the `propagation method` and then clones this method. The cloned `propagation method` can then again provide a cached dispatch node and is able to clone the `propagation method` for its successors and so on.

  The optimization issue will possibly be fixed if initially not all behaviors share the `propagation method`. In this case the question would be which behavior should initially share the `propagation method` and which should not share this method. A starting point would probably be to provide a version of the `propagation method` for each place in the code that creates behaviors.

5 Evaluation

This chapter proposes an evaluation of the results achieved in the development of Reactive Ruby. We initially discuss the design of our evaluation procedure. Subsequently, we explain how the different benchmarks are implemented. Finally, we describe the evaluation of the performance of Reactive Ruby in these benchmarks. The largest part of the evaluation evaluates the peak performance of Reactive Ruby. A discussion of the influence of JIT compilation and the problem of warm-up concludes the chapter.

5.1 Design of the Evaluation

To evaluate Reactive Ruby we chose the Observer design pattern as the base line. We consider the Observer design pattern to be a reasonable choice, since it allows an efficient implementation of reactive applications. In addition, it is still the most common way to implement reactive applications and reactive programming languages aim to replace it. Moreover, in the past the research community used the Observer design pattern to compare the performance of reactive programming libraries in different languages [30].

Another option for the evaluations would be to compare its performance against other efficient reactive languages in Ruby. Unfortunately, very few reactive programming implementations exist in Ruby. Exceptions are early attempts to port ReactiveX to Ruby: RxJRuby [5] and Rx.Rb [6]. The implementation of both of these is in a very early state of development and it is not clear if they are still maintained. Frappuccino [29] is another reactive programming language based on Ruby. However, it is also in an early state of development.

The largest part of the evaluation determines the peak performance of Reactive Ruby. For that, we evaluate its performance in different graph structures after a sufficient amount of warm up. Besides that, we also evaluate the influence the JIT compilation has. For that, the Reactive Ruby runtime system still performs its specialization but no JIT compilation is triggered. This gives an important insight into the necessity of the JIT compilation. Lastly, we evaluate the warm up characteristics of Reactive Ruby, which is a vital property for short running applications.

5.1.1 JavaScript Implementations

We extend the scope of our evaluation to include some benchmarks that focus on JavaScript. The reason for this is that we want to estimate the performance overhead of JavaScript implementations in the perspective of applying the results obtained in Reactive Ruby to JavaScript.

The development status of reactive languages in JavaScript is more advanced than the Ruby counterparts. In JavaScript, reactive programming is widely used to develop reactive web applications. JavaScript offers, among others, Flapjax, ELM and Bacon.js. There is also a well-maintained implementation of ReactiveX, which the industry actively uses.

The evaluation of the performance of certain features across languages is difficult. A direct performance comparison of Reactive Ruby against e.g., RxJS is not that meaningful. There are several variability directions to consider. Different languages use different compilers and we do not want to evaluate the performance of the entire compiler infrastructure. In addition, the base language semantics is different and it is unclear if Ruby or JavaScript is easier or harder to optimize. Also, we only want to evaluate the performance of the reactive features and not the whole system. Therefore, the performance of Reactive Ruby versus the performance of reactive programming libraries in JavaScript is compared indirectly.

For the relative performance evaluation we compare the performance of the different reactive programming languages against the base line in their language. That means we compare Reactive Ruby against an Observer design pattern implementation in Truffle Ruby and we compare reactive programming languages in JavaScript against an Observer design pattern implementation in JavaScript. This solution allows a reasonable relative performance evaluation of Reactive Ruby against reactive programming libraries in JavaScript.

5.1.2 Observer Design Pattern Implementations

The Observer design pattern implementation which is used as the baseline in the benchmarks should provide a fast propagation. That is because the benchmarks measure the time that the propagation of values through a graph structure takes. In more detail, in one iteration of a benchmark several hundred thousand values are propagated through the graph. In addition, the time also includes the creation of the graph structure, however, that is negligible.

The implementation uses arrays to store the observer objects instead of sets or maps which are more common. This is due to the following reasons: we never remove observers in any benchmark. In addition, the Observer design pattern implementation adds observers without checking if they are already added and all observers are always notified in all benchmarks. Therefore, a constant check if an observer is present, a constant removal of an observer and a constant lookup of a certain observer is not needed. Instead, a fast iteration over all observers is desired for which an array provides a better data structure than a set.

In all benchmarks, we used this Observer design pattern implementation to mimic the benchmarks.

5.1.3 Benchmark Designs

We evaluate the performance of Reactive Ruby and two reactive languages in JavaScript in different benchmarks with different graph structures. We provide a chain, a fan and a reverse fan graph structure. In these benchmarks, the performances of the different reactive programming languages is measured in relation to the base line which is an implementation of the Observer design pattern. For that, the benchmarks compare the performance of reactive programming languages in JavaScript against the performance of the Observer design pattern implementation in JavaScript. In addition, they compare the performance of Reactive Ruby against the performance of the Observer design pattern implementation in Ruby.

The chain structure evaluates the performance of behaviors that have one predecessor and one successor behavior. The fan benchmarks evaluate behavior nodes which have a high number of predecessor and successor behaviors. The reverse fan evaluates the impact of a high number of sources which reach one sink behavior.

The chain and the fan are dependency structures that are often used to evaluate the performance of reactive programming [30, 15].

One iteration of a benchmark performs the following: First, it creates a new dependency graph. Then it emits a number of changes from all sources. This has the consequence that every iteration uses new runtime data, however, the structure of the graph remains the same. This means for the JIT compiler that it can reuse code which is based on the structure of the graph, however, that code must be general enough so that it can work with different runtime data.

All evaluations are performed on a Intel Core i7-2640M dual core CPU with 2.8 GHz and 8 GByte ram. The operation system is Ubuntu 14.04. All benchmark evaluations which use Graal and Truffle use the settings -Xmx500m and -Xss2048k.

Peak Performance: In a number of benchmark evaluations we are interested in the peak performance, i.e., the stable state performance. The peak performance is reached after all optimizations have been applied. In technical terms, we do the following to measure the peak performance. The benchmark is repeated until the range relative to the mean of the last N iteration is smaller than a delta. The delta for the benchmarks is 0.2 and the value of N is 10. When the benchmark has reached the peak performance, the median of 10 successive runs is selected for the evaluation.

The Chain Benchmark

The chain benchmark builds a dependency graph composed of a line of nodes. Figure 15 shows the dependency graph. In this benchmark, all behaviors use `map { |x| x + a constant}` as their behavior expressions. The constant is different for each behavior. A chain of nodes is the simplest dependency structure in a dependency graph.

One iteration of the chain benchmark contains the creation of a chain of 10 nodes. After the creation of the chain, 200000 different change events are emitted from the source of the chain.

A chain of nodes is a dependency structure which an efficient reactive programming language needs to optimize well. In this structure, no glitch can occur and therefore, the runtime system does not need to perform checks to ensure glitch freedom. In addition, a behavior in a chain only needs to notify exactly one behavior when it changes.

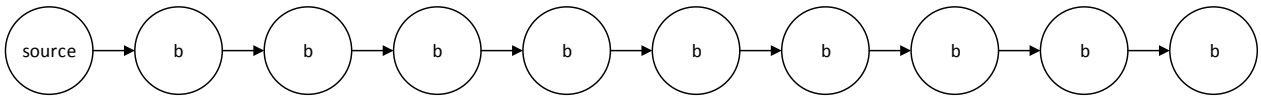

Figure 15: The dependency graph of the chain benchmark.

The Fan Benchmark

The fan benchmark builds a dependency graph in which the nodes in the graph have a high fan-out. Figure 16 visualizes the dependency graph for the fan A and the fan B benchmark.

In the fan A benchmark all nodes except the sink node have 3 successors. All `b` behaviors have the expression `map { |x| x + a constant value }`, however, all of them use a different constant in their expression. The `sink` node in fan A collects the value of all its predecessor nodes. In total, the fan A benchmark contains 14 behavior nodes and one of them is a source node.

In the fan B benchmark, the source node has 10 successor nodes. All `b` behaviors have the expression `map { |x| x + a constant value }`, however all of them use a different constant in their expression. The `sink` node in fan B collects the value of all its predecessor nodes. In total the fan A benchmark contains 12 behaviors and one of them is a source behavior.

In both fan benchmarks one iteration contains the creation of the dependency graph. After the creation of the dependency graph, the benchmark emits 200 000 different change events from the `source` node.

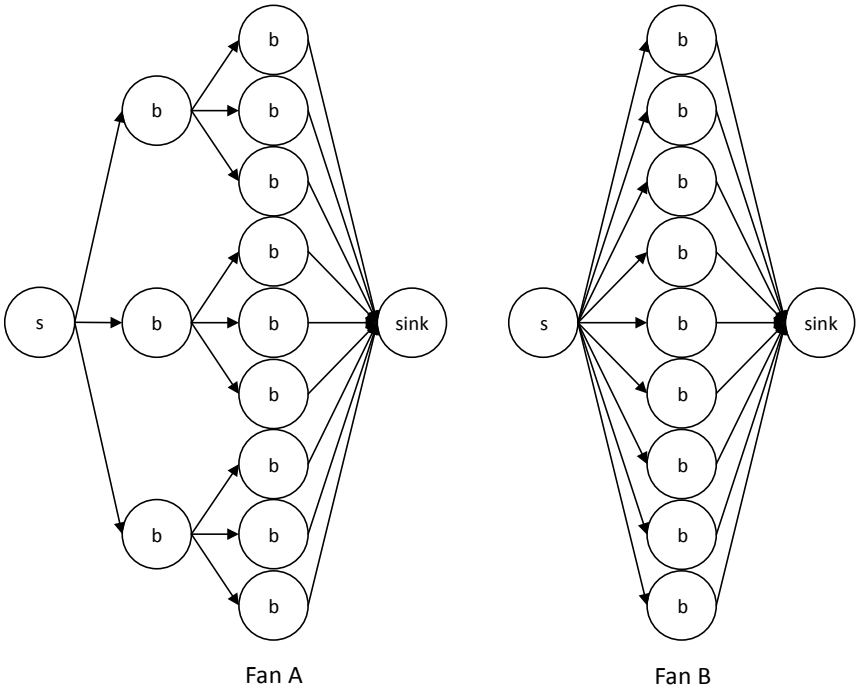

Figure 16: The dependency graph of the fan A and the fan B benchmark.

The Reverse Fan Benchmark

We implemented two reverse fan benchmarks. Both build a dependency graph, which has the structure of a tree. The leaves in that tree are the sources and the root is the sink. Figure 17 shows the dependency graphs.

In the reverse fan A benchmark the dependency graph builds a full binary tree with the sources as leaves. This benchmark contains 15 nodes. Among these 15 nodes 8 nodes are source nodes.

The reverse fan B benchmark builds a chain of behaviors in the dependency graph. The first behavior in the chain is connected to two source behaviors. Every other behavior in this chain is connect to one source behavior. The reverse fan B benchmark contains 15 nodes. Among these 15 nodes 8 nodes are source nodes.

In both reverse fan benchmarks one iteration contains the creation of the dependency graph. After that, the reverse fan A sends 200000 different change events from each source. Since it contains 8 sources, one iteration contains 1600000 change events in total. After that, the reverse fan B sends 25000 different change events from each source. Since it contains 8 sources, one iteration contains 200000 change events in total.

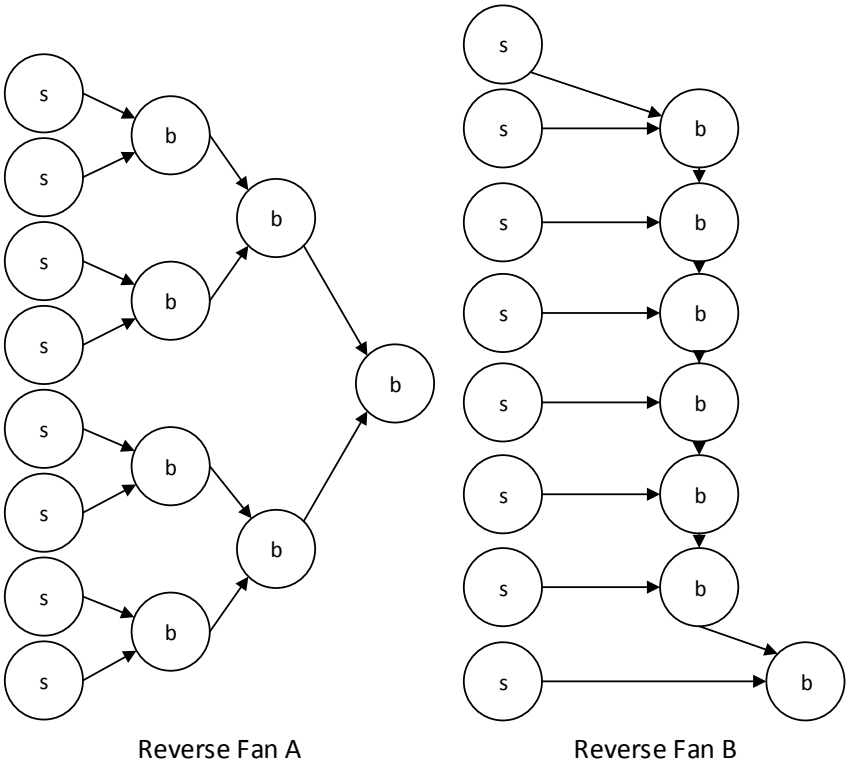

Figure 17: The dependency graph of the reverse fan benchmarks.

5.2 Evaluation of the Peak Performance

The peak performance is an important characteristic. It describes the performance of long running applications. We do not only evaluate the peak performance of Reactive Ruby, we also evaluate the peak performance of RxJS and Bacon.js in JavaScript.

We evaluate the peak performance of Reactive Ruby in a number of benchmarks. All of them share some descriptions, therefore these descriptions are summarized here before we present more details. The plots (figures 18 - 20) visualize the performance of the Observer design pattern base line implementation with the value 1, which represents the performance of the Observer design pattern both in JavaScript and in Ruby. In addition, these plots visualize the relative performance of the different reactive programming languages against the Observer design pattern implementation in its base language. Moreover, all benchmarks share the result that the Graal JIT compiler was able to inline parts of the propagation code at the call site of the `emit` method. In more detail, Graal inlined the propagation code up to the maximum inline code size defined by a parameter. This is a result of an inspection of trace information, which were collected in a separate run.

5.2.1 Chain

Figure 18 visualizes the results of the chain benchmark and table 1 shows the measured execution time.

The performance of Reactive Ruby in the chain benchmark is approximately the same as the performance of the Observer design pattern. Table 1 shows that the difference between the measured execution time of Reactive Ruby and the Observer design pattern baseline is less than 0.01 sec. With the relative performance of ca. 1, Reactive Ruby performs well compared to the reactive programming languages in JavaScript. That is the case since the reactive programming languages in JavaScript are significant slower than the base line. RxJS reaches a relative performance of ca 0.09 and Bacon.js reaches a relative performance of ca 0.03.

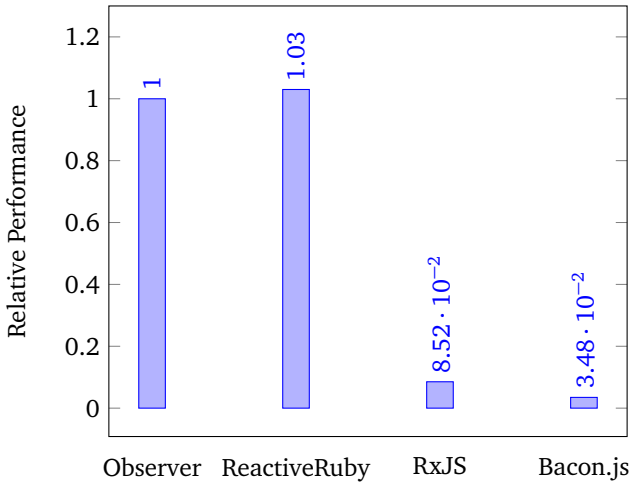

Figure 18: The relative performance of Reactive Ruby, RxJS and Bacon.js compared to the Observer design pattern in the chain benchmarks (higher is better).

Implementation	Runtime in Sec
Observer Ruby	0.02
Reactive Ruby	0.02
Observer JS	0.026
RxJS	0.305
Bacon.js	0.748

Table 1: The execution time of the Observer design pattern implementations and the reactive programming languages for the chain benchmark.

5.2.2 Fan Benchmark

Figure 19 visualizes the results of the two fan benchmarks and tables 2 and 3 shows the execution time measured.

In the fan A benchmark Reactive Ruby reaches a relative performance of around 0.8 and in the fan B benchmark Reactive Ruby only achieves a relative performance of around 0.3. These are the worst relative performances Reactive Ruby achieved in all benchmarks. However, Reactive Ruby still performs better than the reactive programming languages in JavaScript do. Both reactive programming languages in both benchmarks are slower than the Observer design pattern by more than a factor of 10. Therefore, the relative performance of Reactive Ruby is better than the relative performance of reactive programming languages in JavaScript.

In both fan benchmarks RxJS is faster than Bacon.js. In the fan A benchmark RxJS is faster than Bacon.js by more than a factor of 2. In addition, in the fan B benchmark RxJS is faster than bacon.js by almost a factor of 3.

Regarding the plot in figure 19, one could expect that Reactive Ruby performs significantly worse in the fan B benchmark than in the fan A benchmark. That is because its relative performance in the fan A benchmark is 0.82 and in the fan B 0.26, whereas the relative performance of e.g., RxJS is around the same in both benchmarks. However, table 2 and table 3 show that Reactive Ruby performs in both cases very similarly. Truffle Ruby is, however, able to optimize the Observer design pattern implementation for fan B very well. This is also a difference compared to JavaScript. In JavaScript, the Observer design pattern performs very similarly in both fan benchmarks.

Implementation	Runtime in Sec
Observer Ruby	0.08
Reactive Ruby	0.10
ObserverJS	0.068
RxJS	0.847
Bacon.js	1.4745

Table 2: The execution time of the Observer design pattern implementations and the reactive programming languages for the fan A benchmark.

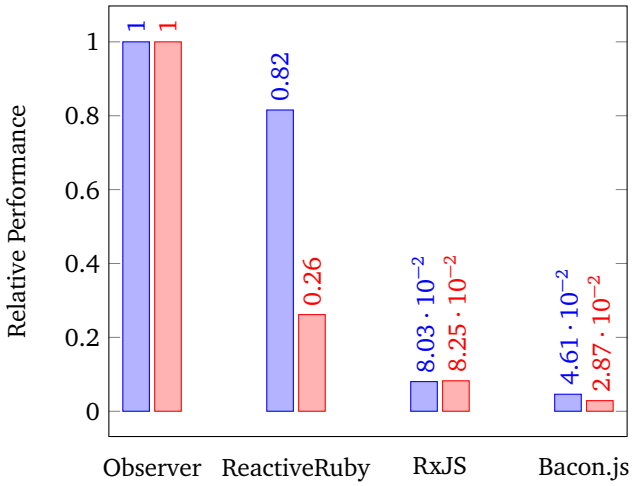

Figure 19: The relative performance of Reactive Ruby, RxJS and Bacon.js compared to the Observer design pattern in the fan A (blue) and fan B (red) benchmark (higher is better).

Implementation	Runtime in Sec
Observer Ruby	0.03
Reactive Ruby	0.11
ObserverJS	0.06
RxJS	0.7275
Bacon.js	2.0885

Table 3: The execution time of the Observer design pattern implementations and the reactive programming languages for the fan B benchmark.

5.2.3 Reverse Fan

Figure 20 visualizes the results of the two reverse fan benchmarks and tables 2 and 3 show the measured execution time.

In both benchmarks Reactive Ruby performs better than the Observer design pattern in Ruby. Tables 4 and 5 show that the difference in execution time between Reactive Ruby and the Observer design pattern is around 0.1 sec for the reverse fan A benchmark and 0.02-0.04 sec for the reverse fan B benchmark. However, we would not claim that our implementation is in general faster than the Observer design pattern, although that Reactive Ruby performs better than the Observer design pattern implementation in these benchmarks.

Reactive Ruby's relative performance is better than the relative performance of RxJS and Bacon.js. The Observer design pattern in JavaScript is faster than RxJS by around a factor of 10. The reactive library Bacon.js performs badly in these benchmarks. The Observer design pattern is faster than bacon.js by more than a factor of 100.

Implementation	Runtime in Sec
Observer Ruby	0.13
Reactive Ruby	0.10
ObserverJS	0.2205
RxJS	2.759
Bacon.js	58.85

Table 4: The execution time of the Observer design pattern implementations and the reactive programming languages for the reverse fan A benchmark.

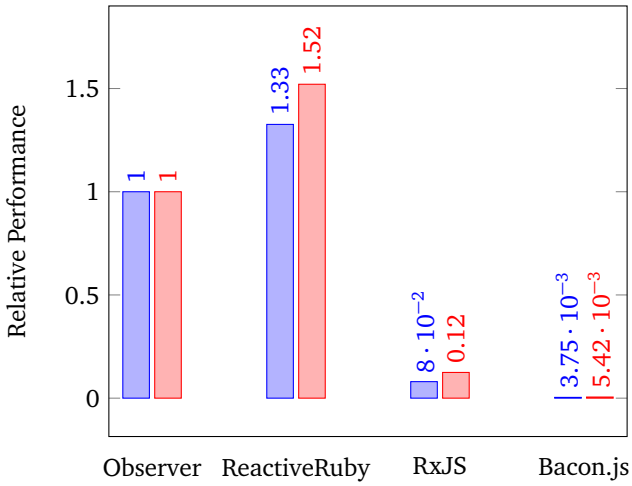

Figure 20: The relative performance of Reactive Ruby, RxJS and Bacon.js compared to the Observer design pattern in the reverse fan A (blue) and reverse fan B (red) benchmark (higher is better).

Implementation	Runtime in Sec
Observer Ruby	0.03
Reactive Ruby	0.02
ObserverJS	0.038
RxJS	0.305
Bacon.js	7.005

Table 5: The execution time of the Observer design pattern implementations and the reactive programming languages for the reverse fan B benchmark.

5.3 Influence of the Compiler

The optimization of Reactive Ruby consists of two parts. These parts are the specializations performed in Truffle and the JIT compilation of the specialized code. The importance of this JIT compilation is evaluated now.

For the evaluation of the importance of JIT compilation, the runtime system of Reactive Ruby still performs all optimizations in Truffle and still uses the Graal VM. That means it e.g., provides the specialized nodes for a chain of behaviors. However, Truffle does not trigger JIT compilation, therefore stable parts of the AST will not be compiled to machine code.

We evaluated the impact of the JIT compilation on the benchmarks chain, fan A and reverse fan A. Figure 21 visualizes the stable performance of Reactive Ruby with and without JIT compilation.

The difference is immense. The average speedup of Reactive Ruby with JIT compared to Reactive Ruby without JIT is bigger than a factor of 50. In the chain benchmarks JIT compilation creates a speedup which is bigger than a factor of 80. In the fan A benchmark JIT compilation creates a speedup of around a factor of 27. In the reverse fan A benchmark JIT compilation creates a speedup of around a factor of 90.

These benchmarks show how important JIT compilation is for the runtime system of Reactive Ruby. Without JIT compilation the performance of the runtime system would be unacceptable in our opinion.

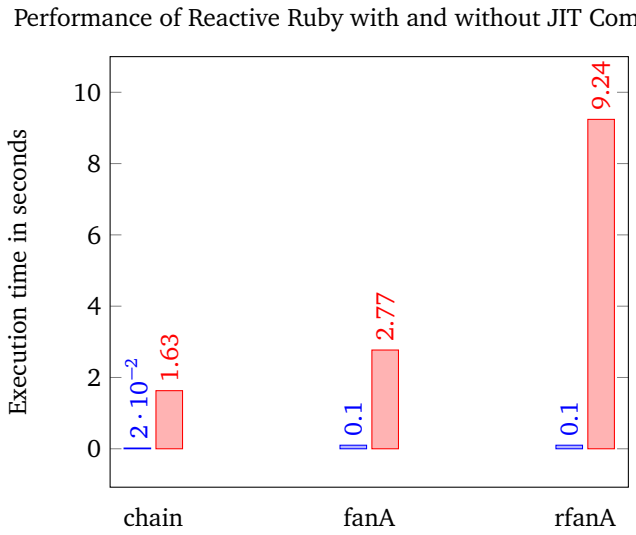

Figure 21: This plot shows the execution time of the benchmarks chain, fan A and reverse fan A. The blue bar shows the execution time with JIT compilation, the red bar shows the execution time without JIT compilation.

5.4 Warm-up Characteristics

Besides the peak performance the warm-up characteristics of Reactive Ruby are interesting. That is because reactive applications are often used for web development where a long warm up is not possible.

Before the warm-up is evaluated, we do a short detour to compile options of Graal. The Truffle / Graal framework has two compilation options. The JIT compilation can be asynchronous in a background thread or synchronous. In general, to gain a better non-warm-up performance the asynchronous compilation is to be preferred. Both compilation options initially slow down the execution of a program when they perform JIT compilation. However, the synchronous compilation slows down the computation significantly more than the asynchronous compilation. Unfortunately, at the point in time when the evaluation was performed, there was a problem with asynchronous compilation of Reactive Ruby. Therefore, the evaluation uses synchronous compilation.

Figure 22 visualizes the execution time of benchmarks over successive iterations. The plot shows the execution time for the chain, fan A and reverse fan A benchmark.

All three benchmarks have a similar warm-up characteristic. The first two iterations take, compared to the compiled code, really long. In particular, they are also significantly slower than the execution of that benchmark in the interpreter (figure 21 for the execution time without JIT compilation). In the third iteration the performance improves significantly. The performance is then in the area of the execution's performance without JIT compilation. In the fourth iteration all benchmarks reach, or almost reach, their peak performance.

The evaluation shows that the performance in the first iteration is by no means perfect. It is actually really slow. The partial evaluation in Truffle with subsequent optimization and JIT compilation in Graal takes time.

The asynchronous JIT compilation improves the performance of the first iterations. However, also with asynchronous JIT compilation, the initial iteration needs significantly more time than the compiled or interpreted code.

The slow first iterations before the runtime system reaches a good performance are not a specific problem of Reactive Ruby. The characteristic of a slow initial performance is also present in Truffle Ruby [40]. Likewise, it is present in the Smalltalk implementation which is implemented in Truffle [32]. The Truffle Graal system in general aims to increase the performance of long running systems.

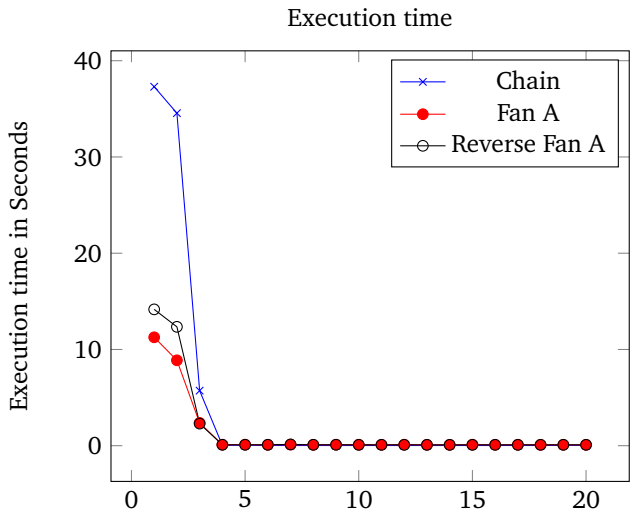

Figure 22: The execution time of the first 20 iterations of Reactive Ruby in the benchmarks chain, fan A and reverse fan A is visualized.

5.5 Conclusion

Let us reconsider the goal for the development of Reactive Ruby which is the reduction of the performance disadvantage of reactive programming compared to the Observer design pattern. In order to reach this goal, Reactive Ruby should combine expressions like `b.map(f).map(g)` during the propagation into one expression. Moreover, it should JIT compile parts of the reactive code.

In the benchmarks Reactive Ruby is able to combine expressions like `b.map(f).map(g)` into one expression during the propagation. However, there is the limitation that Reactive Ruby can only merge this kind of expression starting at the call site of an `emit` method and is limited by the maximum inline size (Truffle parameter). This limitation was not an issue in the benchmarks, however, it could be a problem in other applications.

The evaluation of the peak performance is promising. In the chain and the reverse fan benchmarks Reactive Ruby performs as well as or even slightly better than the Observer design pattern. Especially the result that its performance in the chain benchmark, which models an expression like `b.map(f).map(g)...`, is around the same as the Observer design pattern's performance is important, since chains are a common and unfortunately costly pattern in reactive applications. In the fan benchmarks the performance of Reactive Ruby is still significantly slower than the one of the Observer design pattern. This suggests that Reactive Ruby still needs to improve the implementation of behaviors which have a high fan in and out. In summary, the optimizations of Reactive Ruby reduces the gap between the performance of the Observer design pattern and reactive programming in these benchmarks. Additional evidence for the statement that Reactive Ruby reduces the performance disadvantage of reactive programming is provided by the relative performance comparison to RxJS and Bacon.js, in which Reactive Ruby performs well.

The evaluation clearly states the importance of the JIT compilation which was expected. Nevertheless, the extent of the JIT compilation's influence on the performance is surprising

In short, it can be stated that Reactive Ruby is a promising first step to eliminate the performance gap between reactive programming and the Observer design pattern. However, there are a number of open issues, which we discuss now.

In the evaluation, we compared the performance of Reactive Ruby and the Observer design pattern in the same graph structure. In particular, both graph structures have the same number of nodes. However, in real applications the graph structures (dependency structure) of both of these will probably be different.

This takes us directly to the next issue: The evaluation does not consider real applications. In contrast, it only uses generic graph structures with almost no application logic. Therefore, it is difficult to judge how the evaluation result will transfer to real application.

The evaluation of the warm-up time shows an additional weakness in the Reactive Ruby runtime system. Initially, Reactive Ruby is slow since the runtime system needs time to create efficient code. Once the code is created, it performs well but JIT compilation takes time. This is not a specific Reactive Ruby problem, but a general problem of the approach Truffle/Graal take. However, since reactive programming is often used to develop Web applications which in many cases immediately need a good performance, this can be a problem.

6 Conclusion and Outlook

This chapter concludes the thesis. We summarize the work presented so far and draw conclusions from the evaluation. Finally, we outline future work.

6.1 Summary

This thesis describes the language and the runtime system of Reactive Ruby, a reactive language based on Truffle Ruby. To the best of our knowledge, it is the first reactive language which provides specialized JIT compiler support.

In more detail, Reactive Ruby extends Truffle Ruby. Its runtime system is developed in the Truffle language framework. Therefore, Reactive Ruby is implemented as an AST interpreter which uses node rewriting for local speculative optimizations. Truffle in combination with the Graal VM allows JIT compilations of these optimizations. Since the implementation of Reactive Ruby uses a modified version of the SID UP algorithm as its propagation algorithm, nodes only communicate with their predecessors and successors during the propagation of changes. As a consequence of that, Reactive Ruby can provide local optimizations for the propagation of changes.

Reactive Ruby provides a number of optimizations for reactive programming. Most notably, it is able to combine several behaviors which are close together in the dependency graph during propagation. In particular, it can combine chains of behaviors during propagation. However, there are some limitations regarding the optimization of combining behaviors during propagation.

The evaluation provides evidence that the optimizations of Reactive Ruby reduces the performance gap between reactive programming and the Observer design pattern after warm-up. In a number of benchmarks Reactive Ruby achieves a runtime performance equal to or a slightly better than the Observer design pattern. In particular, it achieves a runtime performance similar to the Observer design pattern in the important chain structure. Additional evidence for a reduction of the performance gap results from the relative performance comparison to RxJS and Bacon.js since Reactive Ruby performs better than both of these in all benchmarks. However, Reactive Ruby is still slower than the Observer design pattern in the benchmarks which evaluated graph structures with a high fan out and fan in. In addition, the benchmarks do not evaluate real applications.

Besides the good peak performance, the evaluation showed that the warm-up characteristics of Reactive Ruby can be a problem. Initially, Reactive Ruby is slow and it takes time until it reaches its peak performance in the benchmarks. This slow warm-up characteristic is not a Reactive Ruby specific problem as it affects other languages implemented with Truffle as well. Nevertheless, it makes Reactive Ruby not well-suited for some domains.

In summary, we conclude that speculative optimizing combined with JIT compilation provides a major chance for reactive programming. These techniques allow attacking the overhead introduced by reactive programming and reduce the performance gap between reactive programming and the Observer design pattern.

We think that further research in this area will probably confirm that these techniques can achieve good results for real applications and allow for faster reactive programming implementations.

6.2 Outlook

The work developed in the thesis opens several opportunities for improvement. Also, in this section we suggest new research areas.

Expressiveness of the Reactive Language

As explained in section 4.7.1 Reactive Ruby is a reactive programming language which neither supports higher-order reactives nor behavior expressions with dynamic dependencies. Therefore, the dependency graph of Reactive Ruby is static. The propagation of changes in a reactive programming language with static dependencies is not as complex as in a reactive programming language with dynamic dependencies. Further research needs to investigate if and how Truffle can provide an efficient runtime system for reactive programming languages with dynamic dependencies. This research contains at least two interesting questions. The first one is about possible optimizations for parts of a dependency graph which very infrequently changes. The other question concerns suitable optimizations for parts of the dependency graph which change on a semi regular basis.

Another restriction of Reactive Ruby is that changes to source nodes must happen mutually exclusive and that during a propagation phase no source is allowed to change. This restriction should be relaxed in the future. Another interesting question is if a form of pipelining during propagation is possible.

Extending the Optimization

Informally said, Reactive Ruby is only able to start the optimization of the change propagation algorithm at points where the imperative code changes the value of a source behavior (section 4.7.1 for more information). Therefore, Reactive Ruby is only able to combine behaviors during the change propagation starting at a source behavior. Thus,

different sources do not share optimizations. Further research should remove these limitations. It is highly desirable that the process of combining several behaviors during propagation is able to start at arbitrary behaviors.

Application and Evaluation

The evaluation of Reactive Ruby provides a first impression of the potentials of Truffle as a runtime system for reactive programming. However, the scope of the evaluation of Reactive Ruby is limited. The evaluation does not evaluate its performance in complex applications. In addition, the performance comparison of Reactive Ruby to other reactive programming languages is across two languages. Further research should evaluate Reactive Ruby's performance in complex and realistic applications. To do so, it would be desirable to have more complex benchmarks as well as non-benchmark applications. In addition, a non-cross language performance evaluation with other reactive programming languages would be desirable.

References

[1] Graal OpenJDK project documentation and Truffle documentation. http://lafo.ssw.uni-linz.ac.at/javadoc/graalvm/all/overview-summary.html and http://lafo.ssw.uni-linz.ac.at/javadoc/truffle/. Accessed: 2015-07-31.

[2] JRuby blog. http://blog.jruby.org/2014/01/truffle_graal_high_performance_backend. Accessed: 2015-07-22.

[3] The Reactive Extensions (Rx). https://msdn.microsoft.com/en-us/data/gg577609. Accessed: 2015-06-11.

[4] REScala web site. https://github.com/guidosalva/REScala.

[5] RxJRuby web site. https://github.com/ReactiveX/RxJRuby.

[6] Rx.rb web site. https://github.com/ReactiveX/Rx.rb.

[7] Truffle Ruby web site. https://github.com/jruby/jruby. Branch truffle-head.

[8] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, August 2013.

[9] Kimberley Burchett, Gregory H. Cooper, and Shriram Krishnamurthi. Lowering: A static optimization technique for transparent functional reactivity. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '07, pages 71–80, New York, NY, USA, 2007. ACM.

[10] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: A declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 178–188, New York, NY, USA, 1987. ACM.

[11] Jay Conrod. A tour of V8: Crankshaft, the optimizing compiler. http://www.jayconrod.com/posts/54/a-tour-of-v8-crankshaft-the-optimizing-compiler. Accessed: 2015-05-19.

[12] Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Proceedings of the 15th European Conference on Programming Languages and Systems*, ESOP'06, pages 294–308, Berlin, Heidelberg, 2006. Springer-Verlag.

[13] Gregory Harold Cooper. *Integrating Dataflow Evaluation into a Practical Higher-order Call-by-value Language*. PhD thesis, Providence, RI, USA, 2008. AAI3335643.

[14] Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 411–422, New York, NY, USA, 2013. ACM.

[15] Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. Distributed REScala: An update algorithm for distributed reactive programming. *SIGPLAN Not.*, 49(10):361–376, October 2014.

[16] Gilles Duboscq, Thomas Würthinger, and Hanspeter Mössenböck. Speculation without regret: Reducing deoptimization meta-data in the graal compiler. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 187–193, New York, NY, USA, 2014. ACM.

[17] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '13, pages 1–10, New York, NY, USA, 2013. ACM.

[18] Jonathan Edwards. Coherent reaction. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 925–932, New York, NY, USA, 2009. ACM.

[19] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997.

[20] Conal M. Elliott. Push-pull functional reactive programming. In *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, pages 25–36, New York, NY, USA, 2009. ACM.

[21] Chris Seaton et al. Core specs report. `http://lafo.ssw.uni-linz.ac.at/graalvm/jruby/specs-core-report/html/`. Accessed: 2015-06-15.

[22] Chris Seaton et al. Language specs report. `http://lafo.ssw.uni-linz.ac.at/graalvm/jruby/specs-language-report/html/`. Accessed: 2015-06-15.

[23] Chris Seaton et al. Stdlib specs report. `http://lafo.ssw.uni-linz.ac.at/graalvm/jruby/specs-library-report/html/`. Accessed: 2015-06-15.

[24] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. *SIGPLAN Not.*, 44(6):465–478, June 2009.

[25] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994.

[26] Vaidas Gasiunas, Lucas Satabin, Mira Mezini, Angel Núñez, and Jacques Noyé. EScala: Modular event-driven object interactions in scala. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, AOSD '11, pages 227–240, New York, NY, USA, 2011. ACM.

[27] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, pages 21–38, London, UK, UK, 1991. Springer-Verlag.

[28] Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. A domain-specific language for building self-optimizing AST interpreters. *SIGPLAN Not.*, 50(3):123–132, September 2014.

[29] Steve Klabnik. Frappuccino web site. `https://github.com/steveklabnik/frappuccino`.

[30] Ingo Maier and Martin Odersky. Deprecating the Observer Pattern with Scala.React. Technical report, 2012.

[31] Ingo Maier and Martin Odersky. Higher-order reactive programming with incremental lists. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 707–731, Berlin, Heidelberg, 2013. Springer-Verlag.

[32] Stefan Marr. Warmup behavior of TruffleSOM and RTruffleSOM. `http://stefan-marr.de/downloads/truffle/warmup.html`. Accessed: 2015-06-15.

[33] Stefan Marr, Chris Seaton, and Stéphane Ducasse. Zero-overhead metaprogramming: Reflection and metaobject protocols fast and without compromises. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 545–554, New York, NY, USA, 2015. ACM.

[34] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A programming language for ajax applications. *SIGPLAN Not.*, 44(10):1–20, October 2009.

[35] Sean Parent. A possible future of software development. `www.stlab.adobe.com/wiki/images/0/0c/Possible_future.pdf`. Accessed: 2015-05-19.

[36] Guido Salvaneschi, Sven Amann, Sebastian Proksch, and Mira Mezini. An empirical study on program comprehension with reactive programming. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 564–575, New York, NY, USA, 2014. ACM.

[37] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. Rescala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 25–36, New York, NY, USA, 2014. ACM.

[38] Guido Salvaneschi and Mira Mezini. Reactive behavior in object-oriented applications: An analysis and a research roadmap. In *Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development*, AOSD '13, pages 37–48, New York, NY, USA, 2013. ACM.

[39] Chris Seaton. Personal web site of Chris Seaton. http://www.chrisseaton.com/rubytruffle/. Accessed: 2015-07-22.

[40] Chris Seaton. Pushing pixels with JRuby+Truffle. http://www.chrisseaton.com/rubytruffle/pushing-pixels/. Accessed: 2015-06-15.

[41] Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. In *International Conference on Functional Programming (ICFP'01)*, 2001.

[42] Zhanyong Wan, Walid Taha, and Paul Hudak. Event-driven frp. In Shriram Krishnamurthi and C.R. Ramakrishnan, editors, *Practical Aspects of Declarative Languages*, volume 2257 of *Lecture Notes in Computer Science*, pages 155–172. Springer Berlin Heidelberg, 2002.

[43] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. An object storage model for the truffle language implementation framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 133–144, New York, NY, USA, 2014. ACM.

[44] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 187–204, New York, NY, USA, 2013. ACM.

[45] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing ast interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, pages 73–82, New York, NY, USA, 2012. ACM.