A Meta Representation for Reactive Dependency Graphs

Eine Meta-Repräsentation für Reaktive Abhängigkeitsgraphen

Master-Thesis von Nico Ritschel

Tag der Einreichung:

1. Gutachten: Prof. Dr.-Ing. Mira Mezini
2. Gutachten: Ragnar Mogk, M. Sc.

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Software Technology Group

A Meta Representation for Reactive Dependency Graphs
Eine Meta-Repräsentation für Reaktive Abhängigkeitsgraphen

Vorgelegte Master-Thesis von Nico Ritschel

1. Gutachten: Prof. Dr.-Ing. Mira Mezini
2. Gutachten: Ragnar Mogk, M. Sc.

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. In der abgegebenen Thesis stimmen die schriftliche und elektronische Fassung überein.

Darmstadt, den 23. Januar 2017

____

(Nico Ritschel)

Abstract

Reactive programming is a paradigm that allows programmers to define perpetual data-flow dependencies in a way that is intuitive and provides an abstraction layer for necessary low-level synchronization operations. The result is a graph-like structure of interconnected operators that process incoming data and transfer it to their successors. However, this structure is typically not accessible to the user of a reactive framework, and may not even be represented by its internal implementation.

In this thesis, we present and evaluate approaches to formally represent reactive data-flow in graph form. We then use approaches known from the field of metaprogramming to create an external meta-representation for reactive programs, which makes the structure and semantics of the data-flow accessible to the user. Additionally, we show how this meta-representation can be transformed into executable code in a reactive framework, and outline how run-time information can be incorporated into the meta-representation graph.

We have implemented the presented meta-representation in the programming language Scala, as well as a transformation procedure into executable code within the REScala framework. Finally, we have evaluated the performance overhead of our implementation and we have implemented several case studies to evaluate the meta-representation's applicability in common metaprogramming use-cases.

Contents

List of Figures

List of Definitions

1 Introduction

In many modern usage scenarios, programs are required to efficiently react to streams of input data and apply complex processing and combination to them in real-time. Simultaneously dealing with common additional challenges like concurrency, data transfer over networks and run-time modifications to the data stream can further complicate this task. While classical object-oriented programming provides solutions for most of these requirements in the form of design patterns and abstraction through libraries, it is often difficult to combine all of them, ensure data consistency and at the same time create code that is efficient and maintainable.

Reactive programming is an approach to these issues that found wide-spread acclaim in research and real-life applications, as it allows programmers to handle the creation, management and usage of data processing streams in an intuitive way that better represents the relevant program flow [1]. With no additional actions required from the programmer, reactive frameworks provide guarantees that are essential to prevent unexpected or unpredictable behavior as well as potential data inconsistencies.

The persistent synchronization of data and the triggering of relevant events based on it is the fundamental principle of reactive programming. This data-flow is commonly visualized as a graph-like network of interconnected operation nodes, and reactive data-flow graphs are often used when analyzing or specifying the semantics of reactive frameworks [1, 2, 3].

Giving users access to the same data-flow modelling can, besides visualization of their programs, allow then an analysis and visualization of their programs. Further, they might be interested in performing transformations on already existing data-flow structures for reasons like optimizations, adaptions to specific execution environments or simply to alter the reactive program's semantics.

The actual implementation of data-flow handling within reactive frameworks may however not reproduce this model but only the resulting behavior, as it is optimized for efficiency while providing an easy to use external interface to the user. These interfaces on the other hand only provide the necessary access for entering or reading data from the reactive network, as well as ways to add additional dependencies. It is therefore not possible to extract and analyze the theoretical graph structure of a reactive program after it is created.

Providing access to the structure of programs as data that can be analyzed and manipulated by other programs is a whole field of software engineering called *metaprogramming*. Metaprograms can take an external view-point that allows them to inspect a *meta representation* of another program or even themselves. This meta representation retains significant semantic information that is not typically available by accessing the internal representation of a program, or simply following the execution of low-level assembly or bytecode.

It is possible to categorize different metaprogramming attempts into *static metaprogramming*, which exclusively inspects a program's structure as it is available at compile-time, and *dynamic metaprogramming* which is applied during run-time and therefore can provide insight into the actual execution behavior of a program under certain input and environment conditions.

In this thesis, after introducing the foundations of reactive programming usage and implementation in Chapter 2, we present a meta representation that is designed to statically model the high-level semantics of data-flow in reactive programs in Chapter 3. We show how this representation can be used to design reactive programs and, in combination with an existing reactive framework, build an executable reactive program in Chapter 4. In this chapter we further show how run-time information can be gathered and re-integrated into the meta representation to analyze the run-time state and behavior of reactive programs.

We have implemented the presented meta representation in the programming language *Scala*, as well as transformations that allows the conversion of reactive programs between this form and an executable representation in the *REScala* framework. In Chapter 5 we present this implementation, outline challenges we encountered and necessary adaptions in comparison to the formal models.

An evaluation of our implementation and our findings can be found in Chapter 6. As part of the evaluation we have applied benchmarking to measure the resulting performance overhead compared to plain REScala code and further evaluated the usefulness for implementing actual reactive metaprograms, by case studies of optimizations and analyses conducted based on the meta representation.

Finally, in Chapter 7 we provide an overview of other research and the resulting approaches in metaprogramming and representation of data-flow. In doing this, we focus on a comparison to our design decisions and potential future extensions that might be applicable to our meta representation.

2 Background

The contributions presented in this thesis are based on reactive programming concepts formed over the last decades that we will briefly introduce in this chapter. Section 2.1 will give an introduction into the relevance and fundamental ideas behind reactive programming. In Section 2.2, we will then present further concepts and terminology that are relevant to the theoretical understanding of the reactive data-flow we model in Chapter 3. Finally, in Section 2.3 we will give an outline about the concrete framework REScala that we used for our implementation presented in Chapter 5.

2.1 Reactive Programming

One of the most basic operations in programming languages is the assignment of a value or an expression to a variable. In purely functional programming languages, this is equivalent to giving a label to the assigned expression allow it to be used multiple times and enable features like recursion. Since purely functional expressions are guaranteed to always evaluate to the same result and have no side effects, it does not matter how often and at which point(s) in the program's execution they are evaluated.

In stateful programming languages however, the evaluation order and number of re-evaluations can be crucial to the program's semantics and therefore need to be exactly specified. The typical approach to define assignments in stateful imperative languages is that assignments represent a one-time transfer of the current expression value at the exact point in time the assignment is executed. By doing so, the assigned expression is evaluated exactly once and its value is stored to allow re-use without re-evaluation.

A consequence of immediate one-time evaluation semantics is that assignments do not establish any permanent dependency between expressions and their assigned variables since later updates of the former don't affect the latter at all. In many common use cases, this behavior is however not the one desired by the programmer. Their intention is instead to establish a constant connection between the source expression and the target value that keeps them synchronized.

To achieve this behavior, there exist multiple workarounds, the simplest being a manual update of the target variable by re-assigning the expression and therefore the updated value. This is obviously not an ideal approach, as it requires the programmer to consider all dependencies whenever a variable is updated, and transitive dependencies between variables can cause long, hardly manageable chains of updates. Especially in modular scenarios where programmers don't necessarily have an overview over the full program code, manually keeping variables synchronized can become tedious, bloats the code with unrelated operations and involves a significant risk of programming errors.

The classical way to avoid manual variable updates in object-oriented programming languages is to encapsulate the variable with and object and its assignment with a method that automatically performs necessary dependency updates. Using a dynamically maintained list of dependencies leads to a paradigm knows as the *Observer pattern* [4] that has become the de-facto standard for variable state synchronization. Allowing dynamic bindings between variables and dependencies however, requires both the observed variable and the dependent observer to adhere to interfaces defined by the pattern, which implement the expected management of dependencies and propagate actual changes.

In Fig. 2.1, examples for both manual updating of dependencies and utilization of a variant of the Observer pattern are shown. While the automated observer code shown on the right side allows a more efficient maintenance of complex data dependencies in the long run, several disadvantages of the observer pattern become apparent:

- The observer management adds a large amount of boilerplate code that is not relevant for the actual computation and makes the intended purpose of the code hard to understand.

```
// Initial setup
var a = 1
var b = a + 1

// First update
a = 2
b = a + 1

// Second update
a = 3
b = a + 1
```

```
// Class definitions
class A(int value) extends Observable {
  private var a = value
  private var observers = mutable.Set()
  def addObserver(o : Observer) = { observers += o }
  def set(value : int) = { a = value; observers.foreach(update) }
}

class B extends Observer {
  private var b = update()
  def update() = { b = a + 1 }
  def get() = b
}

// Initial setup
val a = new A(1)
val b = new B()
a.addObserver(b)

// First update (b is set to 3)
a.set(2)

// Second update (b is set to 4)
a.set(3)
```

Figure 2.1: Non-reactive data dependency handling through manual updating (left) and the Observer pattern (right).

- Only the actual updating process for changes is automatized, but not the actual management of data dependencies, which is still a responsibility of the programmer. This especially becomes problematic when also considering the issue of removing dependencies when they are no longer required, which if forgotten is a common cause for programming errors [5].

- It is technically possible to chain multiple observers triggering each other and create more complex dependency patterns. However, the resulting behavior such as the order or the number of times each variable is updated may depend on the internal definition of the methods defined by the pattern. Managing this process and understanding its consequences requires the user to have insight into the internal control flow, which means that no actual layer of abstraction from the updating process is generated.

Features found in modern programming languages like traits or operator overloading can reduce the visible overhead caused by the Observer pattern, and attempts have been made to further improve the level of abstraction by syntactic abbreviations [6] or macros [7]. They can however not solve the general issue of hard to track control flow and a lack of control over order of the performed updates [5].

Reactive Programming is an programming paradigm designed to replace the Observer pattern by a design that provides the same automatized data-flow as the observer pattern while providing solutions for all the previously listed issues. It has first been developed for usage in purely functional programming languages [8, 9], but its concepts were recently adapted to more commonly used, stateful object-oriented languages [3, 10].

One fundamental element type of reactive programming is *signals*, which represent a variable connected within a dependency network that is automatically updated when the expressions it is based on are changed. In contrast to the observer pattern, all signals in a reactive program are automatically both in the role of a sender of their updated values and a receiver of incoming changes. A special type of signals are *reactive variables*, which act as a *signal source* as they are the only signals not implicitly updated by their dependencies but manually set by the user.

In addition to signals, most reactive frameworks usually also support another type of element that is *events*. Events, while working like signals in receiving, processing and further propagating reactive value changes, do not have a persistent value. Instead, when they are fired they can act as a trigger for reactive signals or other events, and further call *observers*, which are arbitrary functions that can contain external side-effects but are themselves not part of the reactive dependency chain. As with signals, events can exist in both an implicitly triggered form and as an *event source* that is manually fired by the user.

```
// Initial setup
val a = Var(1)
val b = a.map(_ + 1)

// First update (b holds the value 3)
a.set(2)

// Second update (b holds the value 4)
a.set(3)
```

```
// Initial setup
val a = Var(1)
val b = a.map(_ + 1)
val c = b.map(_ * 2)
var d = Signal { b() + c() }
var e = d.changed
e += { v => println("d is now" + v) }

// First update (prints "d is now 7")
a.set(2)

// Second update (prints "d is now 10")
a.set(3)
```

Figure 2.2: Reactive example programs.

The left side of Fig. 2.2 shows a reactive version of the previously shown sample code. The reactive variable *a* is a signal that has the dependent signal *b* added to it. When it is then updated with a new value, changes are implicitly propagated as it would have been the case with the observer pattern, but without any boilerplate code for dependency set-up. The right side of the figure shows a more complex example that creates two dependent signals *b* and *c* for a single reactive variable *a*, which are themselves merged into another signal *d* depending on both. Additionally, an event *e* triggered by the changes to the signal *d* has an observer function added to it that prints the current signal value.

The simple examples shown in Fig. 2.2 already illustrate the significantly reduced boilerplate code needed by reactive programs in comparison to the observer pattern. Another advantage becomes obvious when analyzing the behavior of the program shown on the right side of the figure. In a program using the observer pattern, the observer function might be triggered multiple times after both *b* and *c* are propagating their updates to *d*. The result is multiple printed messages with intermediate values of *d*. The reactive program however avoids this so-called *glitches* that will be further discussed in the next section and only triggers the observer function once after all signals are updated.

This and all further code samples in this chapter are using the syntax of the *REScala* framework for Scala [3]. We will provide a more in-depth introduction to this framework in Section 2.3, and use it as the foundation for our implementation we present in Chapter 5.

2.2 Reactive Data-Flow Management

Both reactive and classical non-reactive programs have in common that they create a chain of operations that process data based on a given control-flow. An established way to visualize this flow in non-reactive programs are *data-flow graphs* [11]. While it is possible to re-use this modelling for reactive programs, data-flow graphs need to be interpreted differently in this context than they would be in their originally specified form. While there each edge only represents an instantaneous one-time transfer of data, for reactive programs each data dependency must be considered as permanent.

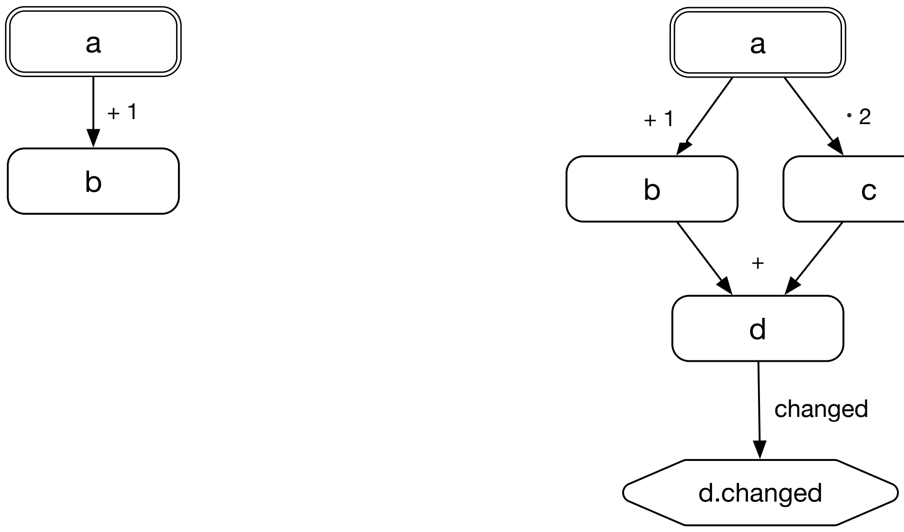


Figure 2.3: Data-flow graphs for the programs from Fig. 2.2.

Fig. 2.3 shows data-flow graphs for the examples from Fig. 2.2. In this thesis, we use rectangular shapes to refer to signals in data-flow graphs and hexagonal ones for events. Event and signal sources are marked by double stroke boxes. While we will re-use this visual design as a guideline for all graph models presented in this thesis, data-flow graphs are only intended to visually support an understanding of reactive data-flow management in this chapter. Therefore, we will also skip all formal specification of data-flow graphs here, which will be discussed in detail in Chapter 3.

To support for the permanent establishment of data dependencies that is the core feature of reactive programming, it is necessary to ensure that data-flow equivalent to the shown graphs is permanently ensured. Compared to alternatives like the observer-pattern, this process takes place completely hidden from the user and is provided by a reactive framework. Two generalized ways to handle data-flow between reactive elements exist in real reactive frameworks [1]:

1. The direction of the given data-flow graph visualization intuitively suggests a propagation of updated values that is *push-based*. In this type of value propagation, reactive updates are pushed from their originating reactive element to its dependencies, and in the same way further on, following the direction of the data-flow graph.

2. An alternative approach that is also found in reactive frameworks is *pull-based* data-flow. In this variant, changes to a reactive element are not immediately propagated but just stored for the element itself. Only if the current value of an element is requested, its incoming data-flow edges are re-evaluated, causing a chain of re-evaluation that proceeds throughout the graph in the opposite direction of the data-flow edges.

While a pull-based management of data-flow is closer to the lazy evaluation semantics used in many functional programming languages like Haskell, it raises the question when re-evaluation of an element must be triggered. In reactive frameworks that only support signals but not events, this trigger is whenever a signal is read. In scenarios where more updates of signals than actual read operations of their value occur, this can lead to performance benefits as value propagation only takes place when the result is relevant.

If events need to be supported on the other hand, a potential triggering of the event's observer functions could take place in every step of the program's execution. Consequently, pull-based approaches can only avoid erratic, delayed triggering of those by permanently pulling for potentially updated values. This behavior typically results in a very low efficiency compared to push-based approaches. For this reason, push-based data-flow management is the more wide-spread variant in real-world implementations of reactive frameworks [1] and will be assumed as the default implementation in this thesis if no explicit management type is stated.

The combination of data-flow graph and used data-flow management semantics is not sufficient to fully define the data-flow of a reactive program. The right data-flow graph shown in Fig. 2.3 is an example where different evaluation orders with different results can exists within a push-based data-flow approach: When the signal source a is updated, an update of both signals b and c is triggered. Assuming a depth-first approach on updating the signals, each of those separately cause the signal d to be updated and the $d.changed$ event to be fired. This means that d is set and the event is fired twice, once with an intermediate value based on an updated b but unmodified c (or vice versa) and again once both b and c are updated.

This behavior may not be the one that a programmer would intuitively expect. Instead, the alternative behavior of updating both b and c first, then d and finally triggering the event $d.changed$ exactly once is the result expected by most programmers. If this assumption is violated by repeatedly causing a signal to re-evaluate or an event to fire based on intermediate values, the resulting behavior is called a *glitch*. An order of propagating value changes that guarantees that each reactive element is updated exactly once is called *glitch-free* [1].

To avoid multiple necessary re-evaluations of a single reactive element, a glitch-free re-evaluation order must ensure that before any signal is updated or any event is fired, all predecessors have already been re-evaluated. An ordering that fulfils this requirement is called a *topological* ordering. There exist multiple approaches to find a topological ordering of a graph, like usage of a priority queue or reverse post-ordering [12]. We will however demonstrate an approach here that is based on level-based node labelling.

In this approach, level labels need to be assigned to each node in a way that they are guaranteed to have a higher level number than all of their predecessors. The most straight-forward method to assign all nodes their correct depth level is to traverse the graph and label each node with the highest level of its predecessors incremented by one. Afterwards a topological ordering is ensured by simply re-evaluating nodes in increasing level order.

Figure 2.4 shows the labeled version of the right data-flow graph from Fig. 2.3 on the left side. Here, the resulting labelling is very simple and intuitive. The right side of the figure shows however an only slightly modified graph with an added direct dependency between the signals b and c. Here, it is significantly harder to intuitively assign labels. The presented approach is however still working as intended and the resulting labels can be seen on the graph.

The issues and design choices shown in this section are a commonality of all reactive frameworks that much establish a permanent data-flow between elements that follows a consistent pattern. There are however additional features that some reactive frameworks implement to extend the feature level of a programming language that can be expressed without leaving the reactive programming context. In the next section, we will present the REScala framework we based our implementation on and the extensions to the presented paradigms and features.

2.3 The REScala Framework

Support for reactive programming has been realized for many different programming languages, both purely functional ones [9] and those already supporting state natively with non-reactive semantics [10]. Assuming a given non-reactive base language, two general approaches for extensions are possible:

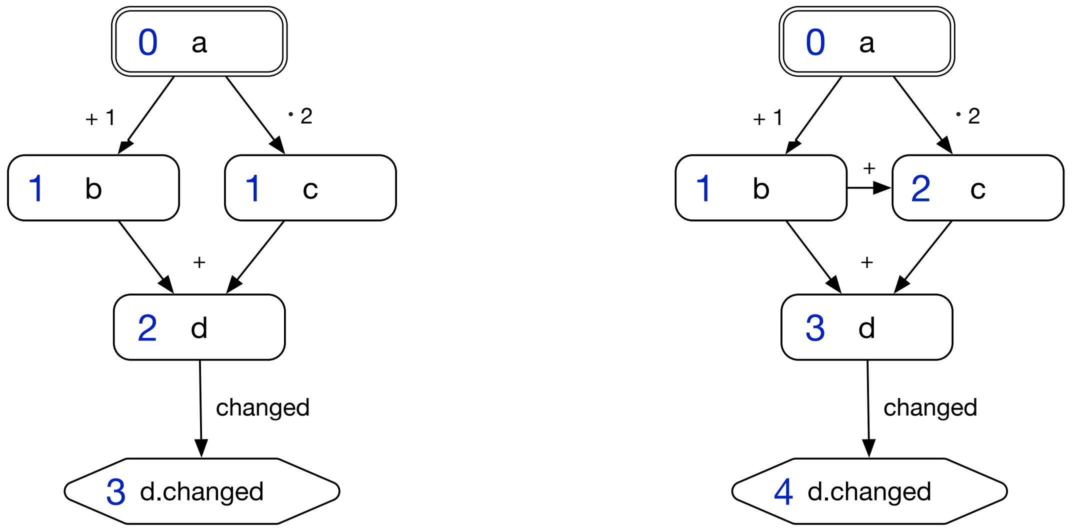


Figure 2.4: Labelled data-flow graph from the right side of Fig. 2.3 (left) and modified, less intuitive version (right).

1. Extending the language with new syntax, either by modifying the original language specification and extending or re-implementing existing compilers.

2. Using the existing syntax and adding support for reactive elements via a library providing the necessary interfaces and background implementation.

The latter approach is usually less invasive and requires less effort of a programmer to set-up and maintain the reactive framework. However, it often comes with the price of less syntactic adaptation to the additional features and more runtime performance overhead compared to language-level extensions.

The programming language Scala [13] provides features to solve both issues: To keep code clean and readable, it provides a highly flexible syntax and expressive language features such as first-class functions, that combined even allow the implementation of many existing language features on a library-level. For performance, Scala code is compiled to Java Bytecode, allowing runtime optimizations through the employed Java VM to further reduce overhead. Because of these advantages, Scala is a language well-suited to implement reactive extensions on a library level.

The REScala framework is an existing library-based implementation of reactive programming features for Scala [3, 14]. It features all the previously described functionality with a push-driven data-flow management model. Signals and events are represented through objects that provide an abstracted interface for the backend propagation functionalities. The processing of events and signals is possible through pre-defined methods based on typical functional operators such as *map* or *fold*, as well as *signal expressions* that allow arbitrary function definitions based on other signals. The examples shown Fig. 2.2 in the previous section demonstrated these basic reactive functionalities.

In extension to the already discussed features of reactive frameworks, REScala supports *dynamic dependencies*. These allow modifications to the statically defined data-flow during a reactive program's runtime. An example usage of dynamic dependencies can be seen in Fig. 2.5, where a higher-order conditional expression is embedded within a signal expression. In a reactive engine that does not support dynamic dependencies, this signal expression would be required to always consider all possible incoming data-flow edges from the signals a, b and c. This would lead to potentially unnecessary re-evaluations of x, for example when c is updated while a is set to *true*. A dynamic dependency model allows the calculation of the actual dependencies of the expression at runtime and therefore can avoid the propagation of updates from the currently not relevant reactive element.

```scala
val a = Var(true)
val b = Var(1)
val c = Var(2)
val x = Signal { if(a()) b() else c() }
println(x.now) // prints 1

c.set(3) // No reevaluation of x
a.set(false) // Changes dependencies of x
c.set(4) // Triggers reevaluation of x

println(x.now) // prints 4
```

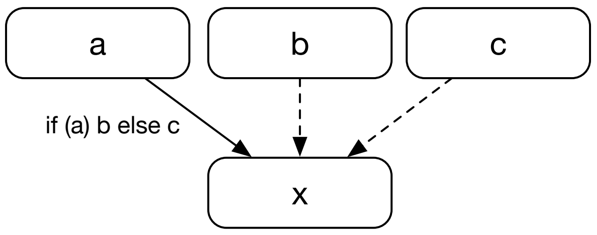


Figure 2.5: Program illustrating dynamic dependencies (left) and resulting data-flow graph (right).

While in the shown example, a dynamic evaluation of dependencies enables a mere optimization of run-time performance, it can also completely enable additional scenarios that would be impossible with purely static dependencies: An operation supported by REScala that would be impossible to handle without dynamic dependencies is the *flattening* of nested reactive values, i.e. where the value of a signal is a signal itself. The result of this operation is a single signal containing the value inner original signal. As the inner signal is dynamically stored within the outer nested signal and may be modified at any time, a static dependency model would not allow to handle this case properly. Instead it would require manual extraction of the outer signal's current value, leaving the reactive context and disabling the provided features and guarantees.

Another feature supported by REScala is the concurrent propagation of multiple updates in the reactive graph. This allows users to access reactive elements from different threads while REScala internally guarantees thread-safety. To allow different implementation approaches to provide this guarantee, REScala provides a well-defined interface and multiple included implementations for *propagation engines* that can be selected as a parameter when using the framework. These engines then control the propagation process and can provide thread synchronization on different granularity levels.

Fig. 2.6 shows a small, simplified excerpt of the interface definitions REScala provides for signals and events. The traits *Signal* and *Event* specify the interface provided to create dependent reactive elements. The subclasses *Var* and *Evt* model the reactive sources that can be manually set or fired by the user. The special method *apply* in line 2 is translated to a simple empty-parentheses call on the signal by Scala and only used within signal expressions to get the current value of the signal while simultaneously creating a dynamic dependency link between the signal expression and the referenced signal.

All interface methods use implicit parameters to allow the selected propagation engine to coordinate the propagation process. The parameter *Engine* used for updating reactive sources specifies the selected engine while the *Ticket* parameter is generated from an engine to ensure that all propagation steps are performed as internally scheduled. As these are internal features of the REScala propagation process that are not relevant for the topics covered by this thesis, we will not further elaborate on their implementation and only show them again when discussing how they affected our implementation in Chapter 5.

```
trait Signal[+A] {
  def apply(implicit ticket: Ticket): A
  def map[B](f: A => B)(implicit ticket: Ticket): Signal[B]
  def changed(implicit ticket: Ticket): Event[A]
}

trait Event[+T] {
  def +=(observer: T => Unit)(implicit ticket: Ticket): Observe
  def map[U](f: T => U)(implicit ticket: Ticket): Event[U]
  def filter (f: T => Boolean)(implicit ticket: Ticket): Event[T]
  def ||[U >: T](other: Event[U])(implicit ticket: Ticket): Event[U]
  def toggle[A](a: Signal[A], b: Signal[A])(implicit ticket: Ticket): Signal[U]
}

class Var[A] extends Signal[A] {
  def set(value: A)(implicit engine: Engine): Unit
}

class Evt[T] extends Event[T] {
  def fire(value: T)(implicit engine: Engine): Unit
}
```

Figure 2.6: Simplified excerpt of the REScala interface definitions for signals and events.

3 A Meta Representation for Static Data-Flow Graphs

The general idea of modelling the statically created data-flow structures within reactive programs through graphs is very common in reactive programming research [1, 2, 3]. Like other authors, we have already used graphical examples for data-flow graphs as in Fig. 2.3 for illustrative purposes in the previous section. Unfortunately, besides outlining this general idea, no efforts have been made yet to also find a common representation for reactive data-flow on a formal level.

To create a meta representation that is suitable for metaprogramming purposes as we intend in this thesis, it is necessary to have an appropriate representation of data-flow first. The most simple approach to define such a representation is to use a generic graph model. Such a modelling is typically defined as a tuple (V, E) of vertices V and edges $E \subseteq (V \times V)$. Applied on reactive programs, this would result in representing each reactive element by a vertex and each data-flow connection by an edge connecting the dependent vertices.

To add information about the direction of data-flow, using directed edges is a measure found in all research, while the used direction of flow is already a design decision. Depending on the desired view-point of the representation, it can be appropriate to consider data-flow as a structural dependency pointing from an element to its source of incoming data-flow, or to follow the direction of the flow as it is propagated from root elements to their destination. In the previous Fig. 2.3 and all following illustrations, we use the latter approach as it avoids a differentiation between edge and flow directions.

While this simple approach is already well-defined model for generic data-flow, it cannot store all of the information contained in graphs as the one shown in Fig. 2.3. Using such a definition, it is neither possible to differentiate between signals and events, nor to clearly identify source elements that can be manually triggered. Furthermore, no semantic information about the operations that are applied when processing the data-flow is made available by this representation.

This lack of a semantic context for the graph's elements makes is difficult to evaluate the data-flow graph in a level of detail that might be necessary for most metaprograms. Besides analyses not being able to access this information, extensions or transformations of existing graphs are almost impossible to perform while ensuring the validity of the resulting graph.

This is because events and signals define different sets of operations that can defined on them in a meaningful way. While it is for example appropriate to filter events, so that they are only fired under certain conditions, signals always need to carry a valid value and make a filtering operation therefore not suitable to be applied to them. Also, operations typically specify exactly which type and number of incoming elements they expect. An example for this can be the *changed*-event as seen in Fig. 2.3, which semantically only allows exactly one incoming signal node as its dependency.

In this chapter, we present more powerful alternatives how to model reactive data-flow graphs. In Section 3.1, we start with a very straight-forward approach of modelling graphs using only two different types of vertices. Then, we show a way to include more information about data-flow semantics in the graph in Section 3.2, and finally in Section 3.3 we add a representation for scheduled updates and retaining the performed operation order.

The order of sections in this chapter follows a design process, starting from a rather straight-forward modelling approach that still shows significant insufficiencies when applied, and moving on to more refined models that solve their predecessor's issues. While this means that the final graph definition as shown in Section 3.3 is the most powerful, this does not necessarily mean that the previous ones, especially as presented in Section 3.2, are not also applicable in situations where for example resources are limited and a simpler model is sufficient.

3.1 Simple Reactive Data-Flow Graphs

As we have outlined, defining a model for data-flow with no distinction between the two fundamental types of elements in reactive programming, signals and events, leads to a significant loss of information. The most intuitive solution is therefore adding a formal separation between these two kinds of vertices in the data-flow graph. This leads to our first formal definition of a reactive data-flow graph.

Definition 3.1 (Reactive Data Flow Graph).

A *reactive data-flow graph* is a tuple (V_S, V_E, E) with

- V_S, the set of signal vertices representing reactive signals,
- V_E, the set of event vertices representing reactive events,
- and $E \in ((V_S \cup V_E) \times (V_S \cup V_E))$, the set of directed data-flow edges between two vertices.

Using the sets V_S and V_E, we can individually handle signals and events in further formal definitions. This allows us to define methods that can be applied to a data-flow graph that allow its extension in a way to ensure a valid data-flow graph. If we start with an initially empty data-flow graph, we can use such methods to formally construct any reactive program in a similar way as a programmer would do in a metaprogram. We will define a sample set of these methods here to allow us the exploration of our model and its expressivity.

We start by defining methods to add signal and event sources to a reactive data-flow graph.

Definition 3.2 (Data Flow Source Construction).

Let $G = (V_S, V_E, E)$ be a reactive data-flow graph, v_S^* be a fresh signal vertex, and v_E^* be a fresh event vertex.

$$G.Var() = (v_S^*, (V_S + v_S^*, V_E, E))$$

$$G.Evt() = (v_E^*, (V_S, V_E + v_E^*, E))$$

The two defined operations *Var()* and *Evt()* add a new signal and event vertex respectively to an existing data-flow graph, and return a pair of both the added vertex and the resulting graph. By repeated application, we can consequently create an arbitrary large graph with source vertices of both types but no data-flow dependencies. Consequently, the next step is to define methods that add dependent signal and event elements based on a graph that already contains at least one of these source elements.

Definition 3.3 (Data Flow Dependency Construction).

Let $G = (V_S, V_E, E)$ be a reactive data-flow graph, $v_S, v_S' \in V_S$ be existing signal vertices, and $v_E, v_E' \in V_E$ be existing event vertices. Further let v_S^* and v_E^* be fresh signal and event vertices.

$$G.changed(v_S) = (v_E^*, (V_S, V_E + v_S^*, E + (v_S, v_E^*)))$$

$$G.mapS(v_S, f) = (v_S^*, (V_S + v_S^*, V_E, E + (v_S, v_S^*)))$$

$$G.mapE(v_E, f) = (v_E^*, (V_S, V_E + v_E^*, E + (v_E, v_E^*)))$$

$$G.filter(v_E, f) = (v_E^*, (V_S, V_E + v_E^*, E + (v_E, v_E^*)))$$

$$G.or(v_E, v_E') = (v_E^*, (V_S, V_E + v_E^*, E + (v_E, v_E^*) + (v_E', v_E^*)))$$

$$G.toggle(v_S, v_S', v_E) = (v_S^*, (V_S + v_S^*, V_E, E + (v_S, v_S^*) + (v_S', v_S^*) + (v_E, v_S^*)))$$

The list of defined methods is exemplary for an arbitrary number of possible operations to connect data-flow vertices. We have selected the ones shown here as a running example, as we will discuss some of them in greater detail in later stages of extending our reactive graph model. However, as all of them represent specific operations applied to a reactive element, we want to give a brief overview of their intended semantics for better understanding:

- *changed* is applied to a signal node and adds an event triggered by the change of this signal.
- *mapS* and *mapE* are applied to a signal or event respectively, and map its value by applying a given function f to it.
- *filter* is applied to an event and filters it based on a given function f. If the function returns true, the filter vertex also fires. Otherwise, the filter vertex ignores the incoming event.
- *or* is applied to two events and fires every time either one of them is fired.
- *toggle* is applied to two signals and an event. Initially, it takes the value of the first signal. Every time the event is fired, it switches to the value of the previously not considered signal.

Using the previous definitions, we can construct or extend data-flow graphs for a large set of reactive programs. While we do not cover all types of vertices that are typically supported by actual reactive frameworks like REScala, adding new types of operations only requires a new method for each of them.

```
$G_0 = (\emptyset, \emptyset, \emptyset)$
$(e1, G_1) = G_0.Evt()$
$(e2, G_2) = G_1.Evt()$
$(v, G_3) = G_2.Var()$
$(or, G_4) = G_3.or(e1, e2)$
$(fil, G_5) = G_4.filter(or, \{\_ > 0\})$
$(mapE, G_6) = G_5.mapE(fil, \{\_ - 1\})$
$(mapS1, G_7) = G_6.mapS(v, \{\_ * 2\})$
$(mapS2, G_8) = G_7.mapS(v, \{\_ * 3\})$
$(tgl, G_9) = G_8.toggle(fil, m1, m2)$
$(ch, G_{10}) = G_9.changed(tgl)$
$(mapS3, G_{11}) = G_{10}.mapS(tgl, \{\_ + 1\})$
```

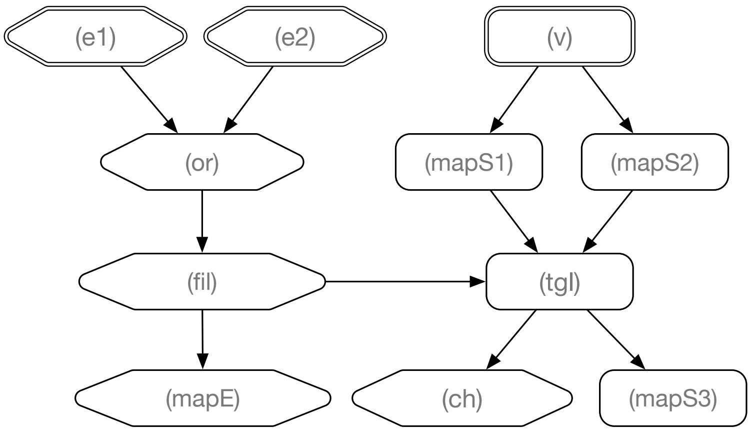


Figure 3.1: Construction of a reactive data-flow graph (left) and visualization of the result G_{11} (right). Node labels are only for visualization purposes and not represented by the graph model.

In Fig. 3.1, we show how to use the presented methods to define a non-trivial example data-flow graph. After defining several source elements using our first set of defined methods, we combine them into new signals and events using the dependency generating methods. The resulting graph models a data-flow as it could occur in a real-life reactive program.

The right side of Fig. 3.1 however also reveals a shortcoming of the presented model for data-flow graphs. While it is now possible to differentiate between reactive signals and events, illustrated as in the previous chapter by rectangular boxes for signals and hexagonal ones for events, all other semantic information, like the mapping or filtering functions f, that is added during graph construction is lost.

Even the labels on the graph's vertices are just added to the figure for illustrative purposes but not retained by the graph representation itself. Consequently, it is, for example, no longer possible to tell the difference between the two mappings *m1* and *m2* of the variable v.

The loss of semantic information however has an even more significant impact: When examining the vertices *fil* and *mapE* in Fig. 3.1, it is no longer possible to tell the difference between event mapping or a filtering. Both vertices are only considered as generic event vertices that process data from an incoming event node. This also becomes evident when comparing their actual definitions, which are completely identical for the presented data-flow graph model.

Due to the significant loss of information, the shown model may not be appropriate for many possible application scenarios, especially when it is used in the context of metaprogramming. Analyses and transformations such as those we will present in Section 6.2 require semantic information to verify if nodes are semantically equivalent, replaceable, or to simply transform existing graphs while retaining the original semantic data used during construction.

3.2 Semantic Reactive Data-Flow Graphs

To overcome the limitations of the data-flow graph model presented in the previous section, it is necessary to store additional semantic information within the graph. A simple idea to achieve this is by adding information to the graph's edges, similar to the visualization shown in Fig. 2.3.

Though this approach works as intended for methods like mapping and filtering, it is not sufficient to fully model the semantics of more complex operations like toggling. Here, a single, semantically atomic operation requires three edges in the graph to be represented. Further, it is necessary to differentiate the edges since the order of the provided signals makes an important semantic difference. A more sophisticated way of storing all the relevant semantic information with the corresponding vertex instead of the edges is therefore necessary.

The underlying task of building a graph annotated with semantic information from many distinct vertex types is also commonly found when building compilers. Here, the most commonly found method of implementing such structures is the construction of *Abstract Syntax Trees (ASTs)* [15]. Syntax trees describe programs by modelling their structure as it is defined by the used programming language, and allow an efficient traversal, analysis and transformation while keeping semantic information easily accessible.

In compilers, ASTs are often implemented in object-oriented programing environments, where different node types are represented by classes using an inheritance hierachy. On a more formal level, they can also be modelled well through *Algebraic Data Types (ADTs)*, which enable the definition of well-defined compound structures that can be nested recursively to build trees. Using ADTs similarly for reactive data-flow graphs, we can create a new definition for *semantic reactive data-flow graphs*.

Definition 3.4 (Semantic Reactive Data Flow Graph).

A semantic reactive data-flow graph $G \subset N$ is a set of vertices that can be constructed from the Algebraic Data Type $N = N_S \cup N_E$.

$$
\begin{aligned}
&\text{Let } N_S^s = \{x \mid x = Var^n && n \in \mathbb{N}\} \\
&\text{Let } N_S = \{x \mid x = MapS(v_S, f) && v_S \in G \cap N_S\} \\
&\qquad \cup \{x \mid x = Toggle(v_S, v_S', v_E) && v_E \in G \cap N_E \quad v_S, v_S' \in G \cap N_S\} \\
&\qquad \cup N_S^s
\end{aligned}
$$

$$
\begin{aligned}
&\text{Let } N_E^s = \{x \mid x = \mathit{Evt}^n && n \in \mathbb{N}\} \\
&\text{Let } N_E = \{x \mid x = \mathit{Changed}(v_S) && v_S \in G \cap N_S\} \\
&\qquad \cup \{x \mid x = \mathit{MapE}(v_E, f) && v_E \in G \cap N_E\} \\
&\qquad \cup \{x \mid x = \mathit{Filter}(v_E, f) && v_E \in G \cap N_E\} \\
&\qquad \cup \{x \mid x = \mathit{Or}(v_E, v_E') && v_E, v_E' \in G \cap N_E\} \\
&\qquad \cup N_E^s
\end{aligned}
$$

Def. 3.4 uses ADTs as a sort of generator for the different types of vertices, which are grouped into four different sets. The set N_S is the set of all possible signals that can exist in a data-flow graph, while the set N_S^s, which is included within N_S, exclusively contains signal sources. The sets N_E and N_E^s use the same pattern for defining all possible events. The actual definition of the semantic graph is simply a subset that contains vertices from all different types of nodes, according to the actually modelled reactive graph. While most graph elements can be simply identified by their structure, source vertices are a special case. Here, we use unique number IDs to allow a distinct addressing.

While we have significantly extended the definition of reactive vertices compared to the one introduced in the previous section, the new semantic graph definition no longer contains an explicit modelling of edges. The reason for this is that edges are now already defined within the internal ADT structure of each element. Nested definitions like $\mathit{Toggle}(\mathit{Var}^1, \mathit{MapS}(\mathit{Var}^2, \{\_ * 2\}, \mathit{Evt}^3)$ can fully represent all dependencies contained within the reactive graph by themselves.

The added semantic information for nodes also allows to verify an already existing graph structure for semantic soundness. While in our first attempt to define graphs, only the domains of the defined methods ensured a valid graph construction, now we can decompose a given graph to validated that it fits the presented ADT generation rules. Yet, we want to also present the updated definitions of the graph construction methods originally introduced in Defs. 3.2 and 3.3.

Definition 3.5 (Semantic Reactive Data Flow Construction).

Let G be a semantic reactive data-flow graph, $v_S, v_S' \in G \cap N_S$ be existing signal vertices, and $n \in \mathbb{N}$ be an index that has not used for any vertex in G before.

$$
\begin{aligned}
&G.\mathit{Var}() = (\mathit{Var}^n, G + \mathit{Var}^n) \\
&G.\mathit{Evt}() = (\mathit{Evt}^n, G + \mathit{Var}^n) \\
&G.\mathit{changed}(v_S) = (\mathit{Changed}(v_S), G + \mathit{Changed}(v_S)) \\
&G.\mathit{mapS}(v_S, f) = (\mathit{MapS}(v_S, f), G + \mathit{MapS}(v_S, f)) \\
&G.\mathit{mapE}(v_E, f) = (\mathit{MapE}(v_E, f), G + \mathit{MapE}(v_E, f)) \\
&G.\mathit{filter}(v_E, f) = (\mathit{Filter}(v_E, f), G + \mathit{Filter}(v_E, f)) \\
&G.\mathit{or}(v_E, v_E') = (\mathit{Or}(v_E, v_E'), G + \mathit{Or}(v_E, v_E')) \\
&G.\mathit{toggle}(v_S, v_S', v_E) = (\mathit{Toggle}(v_S, v_S', v_E), G + \mathit{Toggle}(v_S, v_S', v_E))
\end{aligned}
$$

The new method definitions implicitly use the given ADT generator sets to create add new vertices to the reactive graph. Additional parameters like the mapping or filtering functions f can simply be passed to the new elements. This and the absence of explicit handling for edges makes them more clear than their previous iterations. Also, when comparing definitions like *mapE* and *filter*, they can now be clearly distinguished by their resulting vertex type.

Since the method's signatures were not modified, we can use them to apply the same reactive graph creation process that we have used for illustration in Fig. 3.1. In Fig. 3.2, we show a visualization of

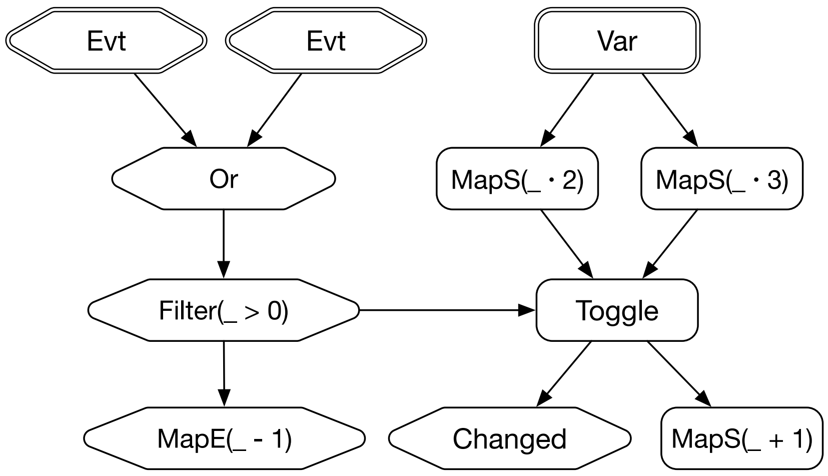


Figure 3.2: Semantic reactive data-flow graph constructed the same way as in Fig. 3.1.

the resulting graph structure with all newly available semantic information. It is now possible to clearly identify every node of the reactive structure and track the applied semantics of the flow.

The chosen approach to represent the graph however also adds new complexity to analysis. To extract the stored information or the implicitly stored data-flow edges, we require a mechanism to decompose each vertex type individually. One approach to handle Algebraic Data Types in programming are partial function definitions that implicitly decompose their parameter via *pattern matching* and one evaluate further where matching succeeds. We imitate this behavior here in our formal definitions by introducing a new syntax we call @-notation.

When a function's parameter is followed by an $@X(y,z)$, the parameter is decomposed and it is checked if it can be generated from the definition of the ADT X. Only if this is the case, the nested parameters of the AST are assigned to the variables y and z that are available like additional parameters in the function's definition body. Otherwise, the function cannot be applied to the given parameter.

The function *dep* we define in Def. 3.6 is an example how to apply the @-notation to process semantic reactive data-flow graphs. It extracts and collects the predecessor vertices that send data-flow to a given vertex. While it primarily serves as an example for pattern matching here, we will re-use it in later sections of this thesis. Implementing individual handling for each node can blow up complexity of other functions that process the graph. Therefore, using helper functions like *def* can greatly improve clarity.

Definition 3.6 (Dependency Collection in the Semantic Reactive Data Flow Graph)**.**

$$dep : G \mapsto \mathbb{P}(G)$$

$$dep(v@Var) = \emptyset$$

$$dep(v@Evt) = \emptyset$$

$$dep(v@Changed(v_S)) = \{v_S\}$$

$$dep(v@MapS(v_S, f)) = \{v_S\}$$

$$dep(v@MapE(v_E, f)) = \{v_E\}$$

$$dep(v@filter(v_E, f)) = \{v_S\}$$

$$dep(v@toggle(v_S, v_S', v_E)) = \{v_S, v_S', v_E\}$$

3.3 Operational Reactive Data-Flow Graphs

The previously presented semantic representation for reactive data-flow graphs models all statically available structural information about a reactive program's data-flow. If we assume a strict two-phase separation of static graph construction and dynamic graph application, all information from the first phase is already sufficiently represented. This assumption may however not hold in all generalized cases, as reactive frameworks typically allow the interleaving of creating new reactive elements and updating the values of already existing ones.

Examples of possible orders in which reactive operations can be executed are shown in Fig. 3.3. Both presented programs have the same graph structure. However, the one on the right side interleaves the graph's construction with its application by changing the value of the source signal a before adding additional dependencies. This interleaving can change the resulting semantics of a program, as only the example on the left fires the event e and therefore triggers printing to the command line.

```
val a = Var(false)
val e = a.changed
val e += println("Event was fired")
a.set(true)
```

```
val a = Var(false)
a.set(true)
val e = a.changed
val e += println("Event was fired")
```

Figure 3.3: Semantically different reactive programs with equal data-flow graphs.

The previous semantic graph model does not capture this difference between the programs as it only contains the graph's structure without any regard for time and order. Therefore, it cannot be appropriately applied to programs like the example on the right side of Fig. 3.3. One way to overcome this limitation would be to consider a fully dynamic representation of the data-flow graph's state, including the results of value propagation and fired events. However, in this section we show an intermediate solution instead, that also solves the shown issue by utilizing only the statically available from a reactive program's code.

To add a representation of time to our graph, we need to transform the previous set of nodes into an ordered list that contains all operations applied to the graph. Besides vertex creation as it is performed by the methods defined in the previous sections, it is necessary to consider all possible updating operations. In our model of data-flow graphs, this is the *setting* of signal sources and the *firing* of event sources. Using these three operation types and the ADTs introduced in the previous section, we can now again create an *operational data-flow graph* definition.

Definition 3.7 (Operational Reactive Data Flow Graph).

An operational reactive data-flow graph is a list of operation log entries $G \subset L$ that can be generated from the Algebraic Data Type E. Let N_S^s, N_E^s, N_S and N_E be defined as for semantic reactive data-flow graphs in Section 3.2.

$$\begin{aligned} \text{Let } L = \{x \mid x = \mathit{Set}(v,x) \qquad & v \in N_S^s\} \\ \cup\ \{x \mid x = \mathit{Fire}(v,x) \qquad & v \in N_E^s\} \\ \cup\ \{x \mid x = \mathit{Create}(v) \qquad & v \in N_S \cup N_E\} \end{aligned}$$

The new definition does no longer contain any set of nodes, but instead an ordered list of operations. Nodes are still represented in the graph as they were in semantic graphs, however embedded within *Create* log entries. Accordingly, it is trivial to update our existing methods for graph construction. Instead of adding new nodes to a set, we simply append a creation entry that contains the node to the graph's log.

Definition 3.8 (Operational Reactive Data Flow Construction).

Let G be an operational reactive data-flow graph, $v_S, v'_S \in N_S$ be existing signal vertices, $v_E, v'_E \in N_E$ be existing event vertices, and $n \in \mathbb{N}$ be an index that has not used for any vertex in G before.

$$G.Var() = (Var^n, G :+ Create(Var^n))$$

$$G.Evt() = (Evt^n, G :+ Create(Evt^n))$$

$$G.changed(v_S) = (Changed(v_S), G :+ Create(Changed(v_S)))$$

$$G.mapS(v_S, f) = (MapS(v_S, f), G :+ Create(MapS(v_S, f)))$$

$$G.mapE(v_E, f) = (MapE(v_E, f), G :+ Create(MapE(v_E, f)))$$

$$G.filter(v_E, f) = (Filter(v_E, f), G :+ Create(Filter(v_E, f)))$$

$$G.or(v_E, v'_E) = (Or(v_E, v'_E), G :+ Create(Or(v_E, v'_E)))$$

$$G.toggle(v_S, v'_S, v_E) = (Toggle(v_S, v'_S, v_E), L :+ Create(Toggle(v_S, v'_S, v_E)))$$

After updating our existing definitions, the only step left to fully represent the different reactive programs in Fig. 3.3 is the addition of new methods that allow the representation of updates. Their definition is very similar to the previous one, as it simply appends a log entry that represent the applied operation.

Definition 3.9 (Operational Reactive Data Flow Updating).

Let G be an operational reactive data-flow graph, $v^s_S \in N^s_S$ be an existing signal source vertex, $v^s_E \in N^s_E$ be an existing event source vertex, and $n \in \mathbb{N}$ be an index that has not used for any vertex in G before.

$$G.Set(v^s_S, x) = L :+ Set(v^s_S, x)$$

$$G.Fire(v^s_E, x) = L :+ Fire(v^s_E, x)$$

$$G.Var(x) = (Var^n, L :+ Create(Var^n) :+ Set(Var^n, x))$$

In our definition, we also introduce a short form for creating variables and immediately initializing them with a certain value. This is a more natural way to set-up variables, that is commonly used in reactive frameworks like REScala and was also used in the code samples from Fig. 3.3. It simply appends two entries to the log, like a consecutive calling of *Var()* and *Set(v^s_S, x)* would do.

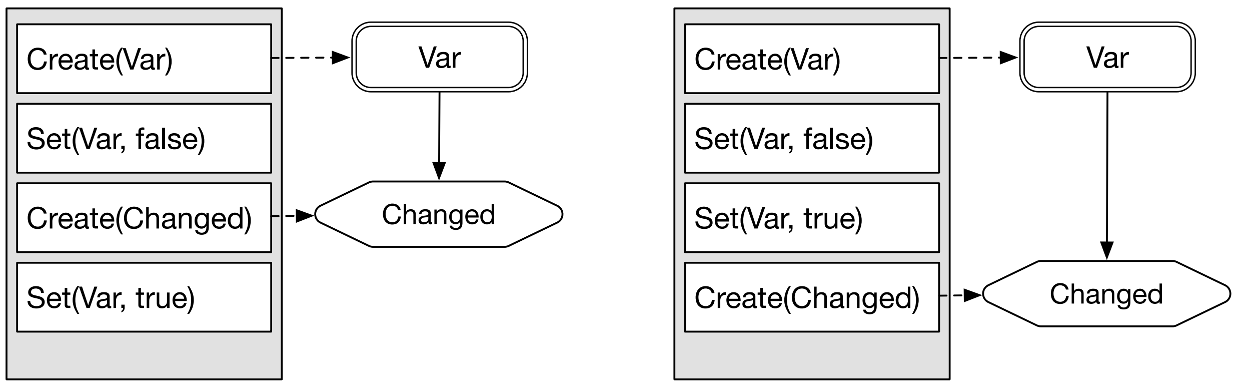


Figure 3.4: Operational data-flow graphs for both of reactive example programs from Fig. 3.3.

Fig. 3.4 shows the operational graphs of the programs from Fig. 3.3 and demonstrates the extended capabilities of the operational graph definition. The log entries on the left side show the difference in operation order, and therefore allow an exact replication of the original program flow. While the operational graph itself only consists of the log on the left side of both examples, we have added the contained graph structure as an additional visualization on the right side. This structure is not lost in the new graph model, but can fully reconstructed from the log entries.

The operational graph definition shown in this section can represent all statically available semantic information found in reactive programs. It is therefore not only suitable to be transformed and dynamically applied as we will present in Chapter 4, but is also suitable for all static analyses and transformations as we will show in detail in Section 6.2. However, for an implementation that enables metaprogramming in combination with a real reactive framework, some additional modifications are necessary in comparison the formal model as it is presented here. We will discuss these in Chapter 5.

4 Dynamic Application of the Reactive Meta Representation

In Chapter 3, we have presented an evolution of data-flow graphs from a very generic model of data-flow dependencies to a complex model of semantics and operation order. What the representation however does not model is the computation for updating reactive values along the data-flow graph. This process leaves the purely static scope we have used in the previous chapter, and instead represents the run-time semantics of a reactive program.

A purely static representation can already be used for certain types metaprogramming. It is however only of limited use without the ability to execute the modelled programs. In the context of applications like optimizations or distributed computation that we will demonstrate in Section 6.2, users typically not only want a formal model of the computed solution, but also an immediate ability to transfer it into code they can execute. In many use cases, this goes so far that all applied metaprogramming steps become transparent to a user writing reactive programs, so they do not have to be involved in them at all.

To make a reactive data-flow graph executable, we have two options that are similar the options when handling of programming languages. First, we can *interpret* the static representation directly and the handle the processing of all applied updates by implementing our own propagation semantics. Second, we can *compile* the representation into a form that can be executed directly or with the support of an external reactive framework. Both approaches have their own advantages and challenges, and many of them are related to the ones arising in classical programming language design [16].

For our work on dynamically applying data-flow graph as we present it in this Chapter, we have focussed on defining a process to transform graphs into an internal representation within a reactive framework that we call *reification*. This way, we can benefit from the existing propagation mechanisms provided by reactive frameworks without having to design these components by ourselves. Consequently, we can treat most implementation details regarding data-flow propagation as a black box.

In reverse, this however also means that we cannot make use of all dynamic information potentially available to the reactive framework, and require specific interfaces for all data we want to gather. We call the process of transforming an already reified reactive program back into a data-flow graph form *unreification*.

The most simple graph model introduced in Section 3.1 is hardly suitable for being reified, as it contains none of the necessary semantic information to create a reactive program from it. Therefore, we start by outlining a generalized reification and unreification process for semantic data-flow graphs in Section 4.1. We then show in Section 4.2 how to adapt this process to apply it to operational data-flow graphs.

Section 4.3 gives a brief outline how dynamic dependencies that are supported certain reactive frameworks can be added to the existing model through reification and unreification. Finally, in Section 4.4 we present an attempt to replace the explicit reification process by an implicit one that transforms data-flow vertices on demand and forms the basis for our later implementation as presented in Chapter 5.

4.1 Reification of Semantic Data-Flow Graphs

On a formal level, it is necessary to differentiate between three different operations that translate between our data-flow graph representation and an executable form. We have already outlined the first two, reification and unreification in the introduction of this chapter. We need however also a third operation, that describes the repeated transfer of a previously reified and unreified reactive graph into a reified form. This operation we call *re-reification*, and it differs from reification as it needs to restore state information that was previously generated for the reified program.

While all transformations can be defined both on a whole-graph level or individually for each reactive vertex, we solely define and focus here on the latter approach as it provides a significantly improved flexibility and allows us the exploration of partially reified graphs, that contain both reified and non-reified vertices. A transformation of the full graph can still be realized through this definition by simply applying it consecutively on all vertices.

Definition 4.1 (Semantic Reactive Data Flow Graph Reification)**.**

Let G, G' be semantic reactive data-flow graphs with $G \subseteq G'$, and $v \in G$ be any vertex of both graphs. Further, let ρ_v be an executable representation of v in the context of G and ρ'_v be the same in the context of G'. We assume further that ρ_v encapsulates an internal state σ_v that can be saved and restored to ρ'_v.

$$\mathit{reify}(v, G) = \rho_v$$

$$\mathit{unreify}(\rho_v, G) = ((v, G), \sigma_v)$$

$$\mathit{rereify}((v, G'), \sigma_v) = \rho'_v$$

The functions in Def. 4.1 are only described in a rather abstract way as the actual process occurring during reification is highly dependent on the used reactive framework. The required capabilities of the framework and limitations, especially when considering multiple different framework implementations for a single reactive graph, are discussed later in Chapter 5 and 6. Then, we will also present a more in-depth description which steps are required to reify a reactive element and how these can be implemented. Similar considerations also lead us to not look in detail at the unreified state information, as it depends on the vertex type and the capabilities of the reactive framework.

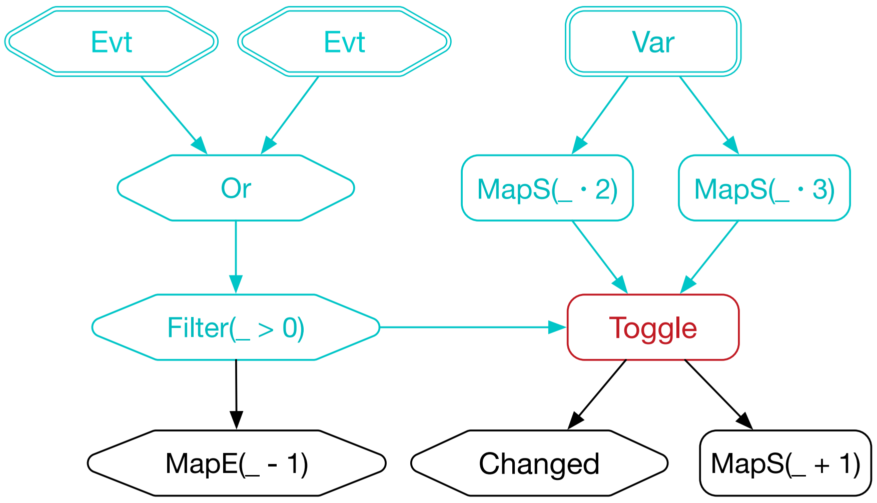


Figure 4.1: Example of recursively descending reification, starting from *Toggle*.

An issue that is independent of the reification process applied to a single vertex is however the approach used to reify necessary dependencies of each element. Naturally, it is not sufficient to simply reify a single reactive vertex by itself, as its whole semantics are determined by its embedding within its graph. This means that reifying a single node must trigger the reification of the nodes it depends on before it can be reified itself. As the same applies to each of these nodes, a recursive chain that requires the reification of a whole subgraph may be triggered. An example of this recursive reification process can be seen in Fig. 4.1 where the reification of the single red *Toggle*-node requires all nodes marked in blue to be reified as well.

To achieve the reified graph to represent the behavior modelled in the meta representation, it is necessary to ensure a persistent binding of reified elements to graph vertices. In the example from Fig. 4.1, this means that after the reification of the *Toggle*-vertex is complete, calling *reify* repeatedly on one the

Var-vertices should not create a fresh, unconnected reactive element but return the same instance that was already created during the recursive reification. Only this way it can be guaranteed that updates of the *Var*-vertex can be propagated properly to *Toggle* by the used framework.

The recursively descending reification of dependencies is however still not sufficient to ensure a consistent state of the reactive graph. In the given graph for example, an update to *Var* would be propagated to *Toggle*, but as the connected vertices *Changed* and *MapS* are not reified yet, they are left out of the propagation process. Since the event is only triggered momentarily, it does not need to be considered for reification. The signal vertex however needs to be updated by the propagation to avoid an inconsistent graph state. Therefore, it also needs to be reified.

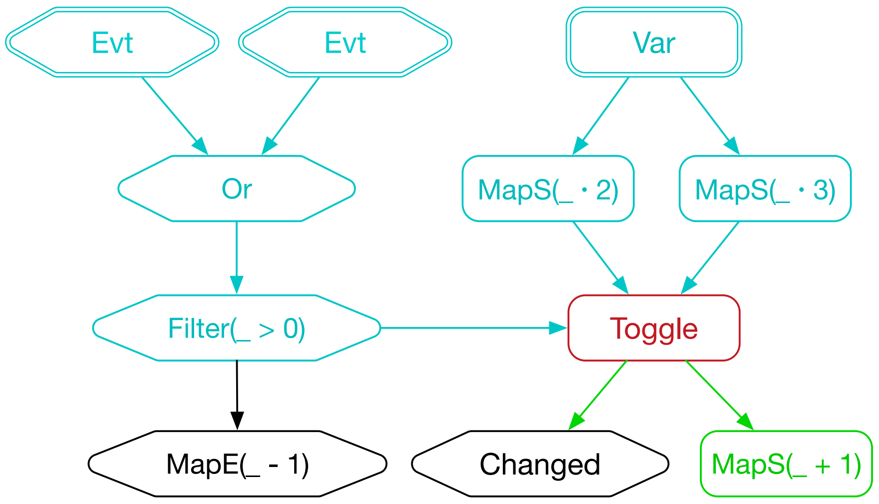


Figure 4.2: Example of recursively descending and ascending reification, starting from *Toggle*.

The required additional reification can be seen in Fig. 4.2, where the result of reifying outgoing vertices is shown in green color. Of course, not only directly depending signals need to be reified, but also signals depending on intermediate events. In the shown example, if there was another signal depending on *Toggle*, it would also need to be reified. Of course, both directions of the recursion interleave each other, so further dependencies of *MapS* in the given example, or *Changed* if it would have further signal dependencies, also need to be reified.

In comparison to reification, the unreification of an existing reactive value is rather simple to schedule. The simplest way to achieve unreification would be to gather all connected vertices, may they either be reified from the same original graph themselves or added dynamically, and create a snapshot of their current state and connectivity. An extension to this approach is to get the original graph as an additional parameter like we do in Def. 4.1, to also add not yet reified or not connected vertices to the graph.

Re-reification of previously unreified vertices allows us to transfer further changes to the data-flow graph to the reified program. Because state information for the re-reified element already exists, this is a naturally more complex task than a blank slate reification of a vertex. It however only affects the implementation level, where it requires the element's previous state to be restored appropriately, even though the surrounding graph structure might have been extended in the mean-time. On the scheduling level we discuss here, re-reification simply requires the same procedure as reification, with the saved state information inserted into all nodes that already had a reification before.

4.2 Reification of Operational Data-Flow Graphs

Being able to transform semantic data-flow graphs into an executable form is a first step into a dynamic application of the reactive meta representation. However, we have already discussed the limitations of this model in Section 3.3, and the inability to model graph updates in the graph is specifically problematic in dynamic scenarios as we present them later in this section. Consequently, enabling the reification and unreification of operational data-flow graphs and the update operations supported by this model is a significant step in extending the dynamic graph capabilities.

A naive way to implement reification of an operational graph would be to simply copy the reification process as shown for semantic data-flow graphs and then applying all relevant setting and firing operations as a separate, second step by simply calling the equivalent operation in the target framework. Unfortunately, as already outlined in Section 3.3, it is essential to maintain the correct order of vertex creation and applying operations to them, and this holds for reifying the graph as well. Simply splitting the reification into two phases would cause a loss of this information and therefore an incorrect final state of the reified graph.

To avoid the loss of operation ordering, it is necessary to access the log information and schedule the reification accordingly. Applying the whole log however has the disadvantage of forcing the reification of potentially irrelevant nodes when only a certain node and its dependencies should be computed. It is therefore desirable to retain the fine-grained per-vertex approach presented in the previous section.

As an alternative to reifying the whole data-flow graph at once, we need to determine the set of nodes that need to be reified, similar to the process we have previously illustrated in Fig. 4.2. However, since applying operations to nodes other than the reified node may also trigger further reifications to be necessary, an iterative approach to find all relevant nodes is necessary.

Definition 4.2 (Computation of Required Dependency Reifications)**.**

```
$\textit{compute-required-reifications}(v, G, V_{checked}) = \{$
  $V_{dep} = \textit{find-all-dependencies}(v)$
  $V_{op} = \{v_0 \in V_{dep} \mid \exists x : (\textit{Fire}(v_0, x) \in G \vee \textit{Set}(v_0, x) \in G)\}$
  $V'_{checked} = V_{checked}$

  foreach $(v_{check} \in V_{op} \setminus V_{checked})$ {
    $V'_{checked} = V'_{checked} + v_{check}$
    $V_{dep} = V_{dep} \cup \textit{compute-required-reifications}(v_{check}, G, V'_{checked})$
  }
  return $V_{dep}$
}
```

In Def. 4.2, we show an algorithm to iteratively compute all necessary reifications based on the list of logs that is the operational data-flow graph. As parameters, it takes the node v that is supposed to be reified, the data-flow graph G and a set of already checked nodes that is initialized as $\emptyset$. It then computes the dependencies V_{dep} of the node v, using a function *find-all-dependencies* that is not shown here but is working based on the approach shown for semantic data-flow graphs. V_{op} collects all of these dependencies that are the target of update operations. Then, the algorithm iterates through all these nodes except those already checked before and performs a recursive call to collect further dependencies for them. Finally, after all nodes with logged operations are checked, the algorithm returns the combined dependency set of all these nodes.

Eventually, the set of nodes returned by the algorithm needs to be reified in order by filtering the log for only entries that affect the collected nodes. This way, the structural approach shown for the semantic graph and the ordered approach for the operational graph can be used in combination to ensure a correct and minimal reification. It should be noted however that the shown algorithm has a rather high complexity itself, and the set of nodes to be reified can become very large. Consequently, it may be more efficient in many cases to simply apply the full log and reify some potentially unnecessary nodes. Only for very large graphs with few operations applied to them, using the shown algorithm for selection would result in an overall computation cost benefit.

Unreifying an existing reactive implementation into an operational data-flow graph is also a more complex procedure if also operations performed directly through the reactive framework need to be considered. While all frameworks need some internal representation of the reactive elements and their dependencies, they typically wouldn't keep a log of the performed operations as their effects are already captured within each element's state information. Consequently, a full unreification would need an extension of the framework to add the necessary logging or a wrapper that forwards operation calls but additionally keeps a protocol. The latter solution will be outlined when considering implicit reification and unreification in Section 4.4.

It could however be argued that the unreified state information is sufficient to keep track of the reactive graph's status. Then, an alternative approach would be to mark the splitting point at which logging has been interrupted through reification and unreification. A later re-reification would then, after normally reifying nodes and restoring their previous state as for semantic graphs, only need to apply operations onto nodes that happened after the last unreification, as previous ones are already reflected in the state.

Definition 4.3 (Operational Reactive Data Flow Graph Reification)**.**

Let G, G' be modified operational reactive data-flow graphs with $G \subseteq G'$ and their list of log entries being either generated members of the original Algebraic Data type L as defined in Def. 3.7, or *Unreify* entries. Let $v \in G$ be any vertex of both graphs.

Further, let ρ_v be an executable representation of v in the context of G and ρ'_v be the same in the context of G'. We assume further that ρ_v encapsulates an internal state σ_v that can be saved and restored to ρ'_v.

$$\mathit{reify}(v, G) = \rho_v$$

$$\mathit{unreify}(\rho_v, G) = ((v, G :+ \mathit{Unreify}), \sigma_v)$$

$$\mathit{rereify}((v, G'), \sigma_v) = \rho''_v$$

Def. 4.3 shows the formalized, abstract definition of reification, unreification and re-reification functions. It uses a special *Unreify* log entry to keep track of the point where the last unreification was applied, and from which point on updates need to be considered during re-reification. This addition is the only significant difference to the previous definition for semantic graphs in Def. 4.1, so the context for the definitions that was given there is still valid.

4.3 Representing Dynamic Dependencies

In Chapter 2.3, we have given a brief introduction into dynamic dependencies in reactive data-flow graphs. Since the original graph definitions shown in Chapter 3 were purely static, it was unnecessary to consider such dependencies as no information about them was available at the time of creating the meta representation graph. This changes when it is possible to unreify already reified and partially executed reactive programs and create a graph representation for them.

So far, we have assumed that the dependency management applied by the reactive framework was also purely static and matched the originally generated graph. If dynamic dependencies are supported and applied for the reifications of nodes, the original static graph may instead be considered to be more of a conservative over-estimation of dependencies. There are however also types of reactive dependencies that cannot be represented statically at all. In Chapter 2.3, we gave the example of flattening a nested signal into a single element that contains the value of the inner original signal. We have illustrated that it is not possible to provide a static overestimation of all possible dependencies of the created *Flatten* node without considering every single other node of the graph.

When the meta representation graphs are however considered as a snapshot of a certain state during run-time, it is possible to represent dynamic dependencies within the existing model. In semantic graphs, this is a rather simple task that may only be limited by the implementation of the unreification process in combination with the reactive framework. If the gathering of dynamic dependencies between elements is supported, they can be treated just like static ones would be, and used in the graph representation.

For operational graphs, the same procedure is possible to unreify an existing reactive graph with dynamic dependencies. Yet, it is important to consider possible dependency changes added by additional operations. In the case of a *Toggle* vertex, this means that a dependency is replaced by another one that is not represented in the current snapshot. For other vertex types like flattening, all known dependency information needs to be considered as lost.

A solution to retain consistency of the graph model could be to separately maintain static and dynamic dependencies and cause a fall-back to the purely static model as soon as an operation is applied to a vertex. Fig. 4.3 shows a sample graph that starts out with only static dependency information but is extended with dynamic information, marked in red, after it has been reified and unreified.

```
$G_0 = (\emptyset, \emptyset, \emptyset)$
$(v1, G_1) = G_0.Var()$
$(v2, G_2) = G_1.Var()$
$(e, G_3) = G_2.Evt()$
$(tgl, G_4) = G_3.toggle(e, v1, v2)$
$((tgl, G_5), \sigma_{tgl}) = unreify(reify(tgl, G_4), G_4)$
```

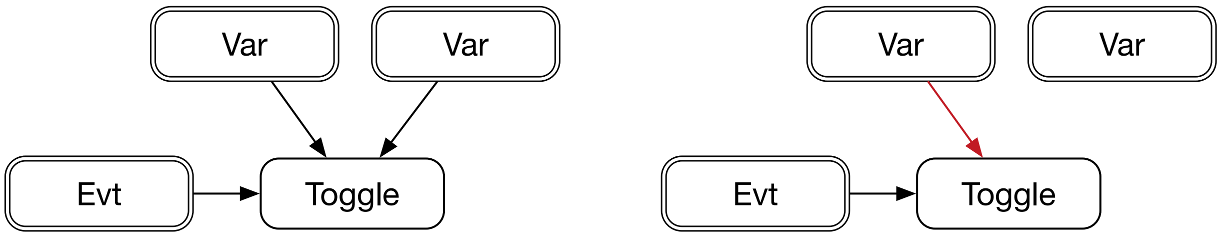


Figure 4.3: Reactive program (top), resulting in data-flow graph with static dependencies (left) and unreified graph with dynamic dependencies (right).

Using a function as shown in Def. 4.2, a further improvement on this approach is possible by computing the set of potentially affected nodes and only partially discarding dynamic information. The result is a mixed graph, that contains dynamic dependencies for some nodes while relying only on static data for other ones. For metaprograms that use dynamic dependency information, for example to generate run-time metrics of a reactive program, such an approach may drastically increase the amount of available run-time data.

While dynamic dependencies in general would also allow node types like the presented *Flatten* to be added to the graph representation, they cause additional issues, even when using the described approaches. The reason for this is that the static fall-back for their dependency set is not defined. To retain the original idea of over-approximating the potential dependencies, edges to all other vertices of the graph would need to be considered, which is hardly applicable for larger data-flow graphs. It is therefore questionable if such a vertex type, even if supported by the used reactive framework, should even be considered within a static graph model.

4.4 Implicit Reification and Unreification

In Section 4.2, we have shown ways to convert between the static operational data-flow graph and an executable program within a reactive framework. This conversion process however required to be triggered manually by the user. While we have ensured that the conversion process is scheduled in a way that always guarantees a consistent state of the program, the user still needs to manage the reified and unreified forms of the reactive program, which causes an undesirable inconvenience.

A much more convenient behavior is an implicit conversion to a reified reactive program. This makes the meta-representation transparent to the user, and allows them to treat it like a regular reactive program if he does not explicitly want to perform manual metaprogramming operations on the graph.

A trivial way to implement an implicit conversion would be to immediately reify all node constructions or updates that are performed by the methods we have presented. We call this approach *eager reification*, was it shares similarities with the program evaluation scheme of the same name. The result would be the graph acting like an extended interface wrapper between the programmer and the reactive framework.

While this still retains some advantages of having a meta representation, such as a unified interface and easier access to the full graph layout, it makes other use cases harder to implement. For example, graph transformations would always require explicit support by the used reactive framework as state information is immediately created for each vertex of the graph and needs to be considered in the transformation. Additionally, using this strategy results in a greater overhead of computation as every operation needs to be mirrored between the both representations, potentially multiple times if dynamic dependencies as described in Section 4.3 are also considered.

A more efficient attempt to implicit reification is to only enforce it when it is necessary. In contrast to eager reification, this approach is similar to the program evaluation scheme of lazy evaluation, which is why we call it *lazy reification*. As in this scheme, the actual reification of vertices is postponed to until a certain *strict point* is reached that requires an immediate, implicit reification of all relevant elements in the graph.

While we have already presented a way to calculate the set of relevant dependencies based on a starting node in Def. 4.2, an implicit approach raises the question how to determine the strict point in an arbitrary sequence of operations. This point is reached in any case when information is required by non-reactive program elements that are not represented in the reactive data-flow graph. Based on the representation presented here and common reactive framework design, there exist two situations where information flows beyond the scope of our graph model:

1. The current value of a reactive signal is queried, for example by an assignment to a non-reactive variable.
2. A fired event triggers an observer function that may perform arbitrary operations based on the propagated value.

Although both situations may seem related at first, there is a fundamental difference between them: A request of a signal's value occurs on the end of the propagation chain and requires a one-time reification and computation of connected elements from bottom-up. It is also possible to consider the registration of an observer for an event as such a request. However, this would require to establish a permanent dependency that requires a constant need to keep the event reified. Alternatively, we can delay the point that forces reification to the time when the observed event is fired for the first time after the observer was added. Only at this point, an actual data-flow is initiated that may trigger the observer and therefore needs to be instantaneously executed.

Handling the first case is consequently more straight-forward than the second one. For value queries, it is sufficient to simply reify the requested element as already shown for explicit reification. The method call then only needs to be forwarded to the appropriate functionality for reading the current value of the executable element within the used reactive framework. The resulting definition is the following one:

Definition 4.4 (Implicit Reification for Momentary Value Queries)**.**

Let G be an operational reactive data-flow graph and v be an existing signal vertex of the graph.

$$G.Now(v) = reify(v, G).Now$$

The case of registered observer however is more complex to handle appropriately. As explained earlier, adding an observer to an event node does not need to trigger immediate reification until the event is fired. Therefore, it is sufficient to simply add an additional logging entry that keeps track of the observer being registered. Additionally, the existing *Set* and *Fire* operations need to be extended to evaluate if a reification is enforced through existing observers.

Definition 4.5 (Implicit Reification for Observed Events)**.**

Let operational reactive data-flow graphs and the sets N_E, N_E^s, N_S^s and L be defined as in Def. 3.7. Let o be a non-reactive observer function and G be a modified operational reactive data-flow graph that has its log entries being generated by either the original Algebraic Data Type L or by L' defined as follows.

$$L' = \{x \mid x = Observe(v_E, o) \qquad v_E \in N_E\}$$

Let $v_E \in N_E$ be an existing event vertex, $v_S^s \in N_S^s$ be an existing signal source vertex and v_E^s be an existing event source vertex.

```
$G.Observe(v_E, o) = G :+ Observe(v_E, o)$
$G.Fire(v_E^s, x)$ = {
    $G' = G :+ Fire(v_E^s, x)$
    $reify\text{-}if\text{-}necessary(v_E^s, G')$
    return $G'$
}
$G.Set(v_S^s, x)$ = {
    $G' = G :+ Set(v_S^s, x)$
    $reify\text{-}if\text{-}necessary(v_S^s, G')$
    return $G'$
}
```

Def. 4.5 first shows the extensions required for supporting observer registration and on-demand reification when events with registered observers may be fired. The addition of the observer log entry, as well as the method to register an observer are very similar to the previously introduced methods for updates. The necessary updates to the methods for firing events and setting signals are also rather minor on a high-level perspective. The only added functionality is the computation of potentially affected observed events and their implicit reification.

The actual computations required to determine these affected vertices are however more complex. First, it is necessary to check the whole chain of vertices depending on the updated reactive element to see if they are events that have any observers attached to them. If any triggered observer is found, it is necessary to reify the triggered event node and all nodes returned by the original dependency computation function from Def. 4.2. This way, it is ensured that the used reactive framework triggers all observers and updates all affected signals correctly.

Definition 4.6 (Computation of Potentially Triggered Observed Events)**.**

```
$reify\text{-}if\text{-}necessary(v, G) = \{$
  if ($has\text{-}triggered\text{-}observers(v, G)$)
    foreach ($v_{dep} \in compute\text{-}required\text{-}reifications(v, G, \emptyset)$)
      $reify(v_{dep}, G)$
$\}$

$has\text{-}triggered\text{-}observers(v, G) = \{$
  $V_{observe} = \{obs \mid Observe(v, obs) \in G\}$
  if ($V_{observe} \neq \emptyset$)
    return true
  $V_{dep} = \{v_0 \in V \mid v \in dep(v_0)\}$
  for ($v_0 \in V_{dep}$)
    if ($has\text{-}triggered\text{-}observers(v_0, G)$)
      return true
  return false
$\}$
```

The resulting algorithm that incorporates the previously outlined steps is shown in Def. 4.6. The function *reify-if-necessary* uses the helper function *has-triggered-observers* to recursively iterate through the outgoing the dependency chain and find event nodes that have a registered observer. If any of them is found, the helper function immediately returns true. This causes *reify-if-necessary* to call the function *compute-required-reifications* that is defined in Def. 4.2, and reify all nodes returned by it.

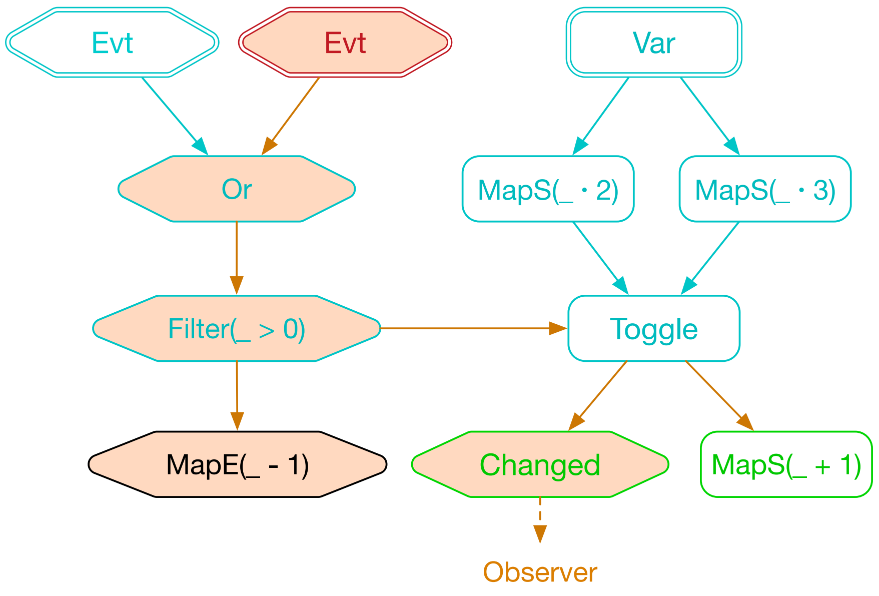


Figure 4.4: Example of implicit reification triggered by firing on indirectly observed event.

Fig. 4.4 shows an example of the presented algorithm applied to a reactive dependency graph. The fired event vertex from which reification originates is marked in red. To check if a reification is necessary, all event nodes along the outgoing data-flow paths need to be recursively checked for observers that may be triggered by the fired event. In the shown example, all event vertices that need to be checked are highlighted with an orange background color. The paths followed by the algorithm are also marked in orange.

The nodes computed to need reification in the second step of the algorithm are marked green and blue as already introduced in Fig. 4.2. While the fired event itself has no descending dependencies, the computed ascending vertices require a reification of all nodes marked in blue. In comparison to this figure, the *Changed*-vertex now also requires reification as it has a registered observer and firing it may therefore have a lasting impact.

The graph demonstrates the fact that, while being as selective as possible, reification can only rely on static information. This means that the actual data-flow may be overestimated, as seen in the given example on the *Filter* vertex. The actual propagation semantics of the value are not considered by the algorithm and only evaluated by the actual reactive framework after reification has taken place. Therefore, even if the fired value would be smaller than zero, the algorithm would still trigger a reification.

When analyzing the outlined lazy reification strategy, an interesting question is how the resulting behavior fits into the binary classification of push- and pull-based reactive propagation schemes as shown in Chapter 2.1. Since an unreified graph only stores performed operations without executing them, the resulting propagation behavior is obviously not a push-based one. In fact, as reification and therefore value propagation only takes place when a value is queried, the approach shown here is more similar to a pull-based propagation scheme.

The important difference is however the handling of events which, as described earlier, is problematic in pull-based frameworks. For them, reification provides a need-based switch from the lazy, pull-based graph to a typically push-based version within a reactive framework that is more efficient and follows the more practical event propagation intuition.

The price of this approach of switching between both the lazy unreified and the eager reified form is however a significant computation overhead when actually implementing the shown algorithms. It consists of finding affected observers for each fired event, computing the set of vertices that need reification and finally performing this reification, potentially resulting in the creation of a full copy of the existing meta-represented graph within the reactive framework. In Section 6.1, we will evaluate how the reification overhead affects the overall performance of constructing and updating a reactive data-flow graph.

5 Implementation of the REScala Meta Representation

In Chapter 3 and 4, we have presented the formal foundations for a meta representation for reactive data-flow graphs. To demonstrate the applicability of the outlined concepts, and in preparation for the evaluation we will present in Chapter 6, we have created an implementation of the meta representation in the Scala programming language. We have further implemented the reification process presented in Chapter 4 for a transformation into the REScala framework that we introduced in Section 2.3.

REScala is well-suited framework to combine with our implementation of the meta representation, as it provides a well-defined interface for creating reactive dependencies on a semantic level. This allowed us to implement most of the desired transformation features without modifying or extending the internal implementation of REScala. While we do not support or make use of all of REScala's features, we were able to experiment with some of them and examine their relation to the presented meta representation.

We start by discussing our approaches to create a reactive interface, similar to the methods introduced in Chapter 3, in In Section 5.1. Our design goal for this interface was that it should be supported by vertices in the data-flow meta representation, and by the reactive elements in the original REScala framework. Section 5.2 then shows our implementation of the data-flow graph meta representation in Scala.

In Section 5.3, we focus on implementing a reification process that resembles the one we have introduced in Chapter 4. Subsection 5.3.1 focuses on the general implementation steps necessary for such a process, independent of the used reactive framework. We then discuss in Subsection 5.3.2 how this concepts can be implemented on the concrete example of REScala.

By splitting our implementation into individual segments, most of which are independent from the used reactive framework, we try to demonstrate that our design is not tailored specifically to REScala but universally applicable. Unfortunately, we did not have the chance to implement further reifications for other reactive frameworks in the scope of this thesis. We are however confident that our design is flexible enough to support a wide range of reactive framework alternatives.

5.1 A Common Interface for Typed Reactive Graphs in Scala

When we introduced the semantic data-flow graph representation in Chapter 3, we made use of Algebraic Data Types in our definitions. This way to model a structural vertex hierarchy is an ideal fit for an implementation in the programming language Scala, as it supports *case classes* and *pattern matching*, which allow us to transfer our definitions to program code in a very straight-forward way.. Additional features like object-oriented subtyping and inheritance, as well as an existing first-class representation functions in Scala further improve the direct transformability of the formal model to a functional implementation.

A feature that was not taken into account in Chapter 3 is type-safety for data that is contained and propagated by signal and event contents. In a language with a strong static type system that supports generic types like Scala, it only makes sense to provide full support for reactive content with type parameters. Another dimension of subtyping that should be taken into regard is the actual implementation used for internal handling by a reactive framework. Here it is most desirable to provide a unified interface of supported methods for the graph construction or updating as defined in Chapter 3, that is implemented by both our meta representation and reactive frameworks like REScala.

Figure 5.1 shows three different dimensions of subtyping enabled by an interface definition as originally shown for REScala in Fig. 2.6. In this example, the *base* variable can be assigned of all three following values as all reactive variables are also signals, signals are defined as covariant in their content type and the common reactive interface is a supertype of all of its implementations, here represented by a sample implementation *SignalImpl*.

```
class B extends A
class SignalImpl[+A] extends Signal[A] { ... }

val base : Signal[A]
val sub1 : Var[A]
val sub2 : Signal[B]
val sub3 : SignalImpl[A]
```

Figure 5.1: Three dimensions of reactive subtyping in Scala.

Although it seems that the existing interface definition for REScala already supports all these subtyping dimensions in a reasonable way, it does not take into account an important restriction that is necessary for practical implementations of signals and events. While all implementations should implement the same interface, they should still be distinct enough so that their methods don't allow a mixing of different implementations within the same data-flow graph. In the concrete example from Fig. 5.1, this means that vertices of type *SignalImpl* should not be allowed to be combined with other implementations by using method calls of the signal interface.

Our goal is therefore to define an interface that is unified and allows an efficient replacement of one used reactive element implementation by another one, while simultaneously specifying that methods only accept other vertices of the same type of implementation. Most object-oriented programming languages don't provide support for such restrictions when defining class interfaces. Scala's type system however has a feature that enables such a feature: *F-bounded types* allow a class or interface to access the implemented subtype through a type parameter while higher-order types allow these subtypes to be parametrized themselves [17]. These two features can be combined to create a type-safe cycle of dependencies that guarantees that only compatible implementations of the signal and event interface can be mixed when creating a new reactive element.

```
trait Event[+T,SL[+X] <: Signal[X,SL,EV], EV[+Z] <: Event[Z,SL,EV]] {
  this: EV =>

  def ||[U >: T](other: EV[U])(implicit ticket: Ticket): EV[U]
   ...
}
class EventImpl[+T] extends Event[T, SignalImpl, EventImpl] with ImplSpecifics[T] {
  override def ||[U >: T](other: EventImpl[U])(implicit ticket: Ticket): EventImpl[U]
 ...
}
trait Evt[T,SL[+X] <: Signal[X,SL,EV], EV[+Z] <: Event[Z,SL,EV]] extends Event[T,SL,EV] {
 ...
}
class EvtImpl[T] extends Evt[T, SignalImpl, EventImpl] with ImplSpecifics[T] {
 ...
}
```

Figure 5.2: Event interface definition using cyclic F-bounded higher types.

Fig. 5.2 shows the resulting definitions for the generic *Event* and *Evt* interfaces, the latter representing event sources. It also shows the required definitions for implementations for both interfaces. The type parameter *T* is the content type of the event, while *SL* is the supported signal and *EV* the supported event type for mutual combination. In the actual implementation, these type parameters are filled with the implementation subtypes. The definition of the *this*-type through the interface ensures that only the correct subtype may be used in the definition. The *Evt*-interface mirrors the event interface definition pattern while also establishing the subtype relationship with generic events.

While the implementation shown above works as intended, it drastically increases the complexity of the used type definitions and requires far more in-depth knowledge about Scala's type system than the original definition that was presented in Fig. 2.6. Furthermore, it reduces the code's readability by adding type information that is not directly related to the code's functionality. Scala's type inference and direct usage of implementations instead of the generic interface can hide most of the complexity from the user. This is however in contrast to the supposed benefit of a unified interface to provide independence from a concrete implementation. Therefore, the shown implementation is not very suitable for practical usage by an external programmer.

An alternative way to implement subtyping that is more common in functional languages like Haskell are *type classes*. Since Scala as an object-oriented language with functional elements also allows Haskell-style type declarations and ad-hoc polymorphism, it provides the necessary means for implementing this approach. The result is an interface that provides a bridge between a functional and an object-oriented design style and is different in its presentation than the one previously shown.

```
trait Api {
  type Signal[+A]
  type Event[+A]
  type Var[A] <: Signal[A]
  type Evt[A] <: Event[A]

  def Evt[A](): Evt[A]
  def Var[A](v: A): Var[A]

  def set[A](vr: Var[A], value: A): Unit
  def or[A, B >: A](event: Event[A], other: Event[B]): Event[B]
   ...
}

object ApiImpl extends Api {
  override type Signal[+A] = SignalImpl[A]
  override type Event[+A] = EventImpl[A]
  override type Var[A] = VarImpl[A]
  override type Evt[A] = EvtImpl[A]

  override def Evt[A](): Evt[A] = new EvtImpl()
  override def Var[A](v: A): Var[A] = new VarImpl(v)

  override def set[A](vr: Var[A], value: A): Unit = vr.set(value)
  override def or[A, B >: A](event: Event[A], other: Event[B]): Event[B] = event.or(other)
   ...
}
```

Figure 5.3: Type class interface for reactive operations.

An excerpt of a type class interface for reactive operations implemented in Scala can be seen in Fig. 5.3. The abstract *Api* trait defines the operations that have to be supported by a reactive programming implementation while the concrete object *ApiImpl* provides the necessary binding to the actual functionalities provided by a reactive graph.

To see the advantage of using type classes over direct inheritance as shown in Fig. 5.2, it is important to understand that in this version, *SignalImpl* and all potential other implementations can be completely independent and don't need any common superclass or interface. This provides greater flexibility for each implementation while also implying the limits of interoperability that had to be defined explicitly and with a high price of complexity in the pure object-oriented interface.

The main drawback of the type class interface is the necessity to mix two syntactically very different programming styles. Especially when the concrete implementation to be used for a program is know, this can result in calls as shown in Fig. 5.4, that all construct the same graph but range from a more functional to a more object-oriented style.

```
val e1 = ApiImpl.mapE(ApiImpl.or(ApiImpl.Evt(), ApiImpl.Evt()), f)
val e2 = ApiImpl.mapE(ApiImpl.Evt().or(ApiImpl.Evt()))
val e3 = ApiImpl.Evt().or(ApiImpl.Evt()).mapE(f)
```

Figure 5.4: Three different ways to create the same event chain.

Since this ambiguity is only a minor issue that can be avoided by users by deciding for a consistent programming style, while the type class implementation achieves a greatly improved interface clarity and usability, it is the variant we decided for when designing the actual meta representation for reactive graphs.

5.2 Implementation of the Meta Representation in Scala

With a well-defined common interface for reactive graph nodes as presented, it is possible to create the structures used by the formal definitions shown in Chapter 3 in Scala. The final version of the Scala implementation created for this thesis is based on the fully-featured operational reactive data-flow graph presented in Section 3.3. The other presented representations can be considered subsets of this implementation that are not explicitly shown here but could be derived from the final expansion stage presented here without much additional work being required.

Since the operational graph meta representation not only uses a set of nodes to define its graph structure but also an added log of operations, it makes sense to represent the graph itself as an object collecting this information and making it accessible in one place. Fig. 5.5 shows the core definition of a reactive data-flow graph in Scala that is based on the stateful graph definitions from Section 3.3 with a collection of nodes and a list of log entries.

Fitting the theoretical definitions shown in Chapter 3 into an actual implementation leads to some adjustments that become apparent in Fig. 5.5. Most significantly, the actual graph implementation is mutable, so that constructing new elements does not result in an independent copy being generated but in a modification of the existing graph. To avoid a loss of abstraction by giving the user direct access to the internally used mutual data structures, accessing members externally generates a version of each collection that is immutable, while mutations are only possible through interface functions like the ones for adding elements shown in Fig. 5.5, that provide sanity checks and limit the available operations.

Also related to mutability is the notable extension of the graph by *node references* as an additional component as seen in Fig. 5.5. References act as a layer of indirection, similar to object references in programming languages like Scala. This means that multiple references pointing to the same node can exist, and it is further possible to redirect an existing reference to another node without replacing the reference instance.

The rationale behind adding references is that they allow an efficient implementation of graph mutations, like the merging of two reactive nodes into one, or the replacement of existing nodes. This type

```
class ReactiveGraph {
  def nodes: Set[ReactiveNode[_]]
  def refs: Map[ReactiveRef[_], ReactiveNode[_]]
  def log: List [ReactiveLog[_]]

  def registerNode(ReactiveNode[_]): Unit = ...
  def registerRef [T](ReactiveRef[T], ReactiveNode[T]): Unit = ...
  def deref[T](ReactiveRef[T]): ReactiveNode[T] = ...
  def addLog(ReactiveLog[_]): Unit = ...
}
```

Figure 5.5: Definition of the reactive data-flow graph class in Scala.

of operations is heavily used in our case studies about graph optimization as presented in Section 6.2.1. With references, it can be ensured that the results of such operations are correctly mirrored into user code pointing to a specific node. Through the reference count for nodes that also becomes available this way, it is further possible for optimizations to ensure that no undefined references, similar to *null*-references in Scala and other programming languages, are created.

The decision to use a mutable graph representation is mostly motivated by practical reasons, as it simplifies reification to the REScala framework and enables an easier approach to implicit reification. A disadvantage of the chosed implementation design is however the loss of internal type-safely because of the limitations of Scala's type system: When storing objects with unrestricted type parameters in collections, this parameter of the individual object is lost. In the shown code example, this means that especially the *refs*-map cannot implicitly ensure that keys and values have matching type parameters. The encapsulating methods can however provide a reasonable compensation for this by ensuring type-safety for external users and avoid runtime errors that could theoretically occur when using a method such as *deref* that internally has to perform an explicit type-cast.

```
trait ReactiveNode[+T]

trait SignalNode[+A] extends ReactiveNode[A]
case class VarSignalNode[A](g: ReactiveGraph) extends SignalNode[A]
case class MappedSignalNode[A,+B](g: ReactiveGraph, base: SignalRef[A], m: A => B)
    extends SignalNode[B]
case class ToggledSignalNode[+T,+A](g: ReactiveGraph, base: EventRef[T], a: SignalRef[A],
    b: SignalRef[A]) extends SignalNode[A]

trait EventNode[+T] extends ReactiveNode[T]
case class EvtEventNode[T](g: ReactiveGraph) extends EventNode[T]
case class ChangedEventNode[+T](g: ReactiveGraph, base: SignalRef[T]) extends EventNode[T]
case class MappedEventNode[T, +U](g: ReactiveGraph, base: EventRef[T], m: T => U)
    extends EventNode[U]
case class FilteredEventNode[T, +U >: T](g: ReactiveGraph, base: EventRef[T],
    pred: T => Boolean) extends EventNode[U]
```

Figure 5.6: Definition of the reactive data-flow vertex structure through case classes in Scala.

After defining the data-flow graph as whole, we need to translate the Algebraic Data Type structures formally defined in Chapter 3 into Scala code. The code in Fig. 5.6 shows the basic class structure used to represent the different node types as case classes. We define base traits that are separate for events and signals, but shared for all node types. This is the code equivalent for defining two separate sets of vertices and methods that have only one of them as their domain, as in Def. 3.4 and 3.7. The actual case class definitions are almost equivalent to the ADT generators introduced in the same definitions, except that they have the data-flow graph as an additional parameter, which allows them to register themselves with it.

All references between nodes in the created tree structures are using references instead of direct links to nodes, which allows them to be redirected by optimizations if necessary, just like external references. While all node types have in common that they carry a type that describes their represented content, many of them are actually generic in more than one type parameter. For the *MappedEventNode* class for example, both the incoming base node's content type and the represented mapping function's parameter and result type are independent parameters that have to match correctly and are also used to define the node's own content type.

The content type is further defined to be covariant, which allows subtyping as shown in the variable *sub2* in Fig. 5.1 and satisfies the type requirements of the type class interface introduced in Fig. 5.3.

```
trait  SignalNode[+A] extends ReactiveNode[A] {
  g.registerNode(this)
  g.addLog(LoggedCreate(new SignalRef(this)))

  def changed: ChangedEventNode[A] = ChangedEventNode(g, new SignalRef(this))
  def map[X >: A, B](f: (X) => B): MappedSignalNode[X, B] = MappedSignalNode(g,
       new SignalRef(this), f)
}

case  class  VarSignalNode[A](g: ReactiveGraph) extends SignalNode[A] {
  def  set(value : A) : Unit = g.addLog(LoggedSet(new SingalRef(this), value))
}
```

Figure 5.7: Excerpt of the meta representation implementation for signals and signal sources in Scala.

Supplementary to the basic definitions in Fig. 5.6, Fig. 5.7 gives a more in-detail look into the inner workings of the classes and the methods provided by the signal nodes to allow data-flow graph construction. The first two operations of registering the node into the graph and adding a log entry are performed immediately when the node is constructed.

The exemplary interface methods then allow to create new dependent nodes by creating a new node referencing the one whose interface is called. While the introduction of the additional type parameter *X* in this example may seem unnecessary at first, it is required as not both type parameters of *MappedEventNode* are defined as covariant. In Scala's type system it is not directly possible to interchange invariant and covariant type variables, which makes in necessary to use an additional intermediate type that is lower-bounded by the covariant *A*. Similar workarounds are needed for some other node types and operations to make them type-safe in the context of the type system provided by Scala.

The class representing signal sources adds an additional method to the interface to allow the source vertices to be set. This is implemented by adding a log entry to the graph. The class definitions for log entries are not shown here as they don't implement special functionality but simply provide a structure similar to the formal definition in Def. 3.7. Implementation details for references are also not shown here as they simply provide the ability to be de-referenced through the data-flow graph's mapping functionality. They however also mirror the same operations to provide further convenience. This allows the meta-implementation of the type-class interface from Fig. 5.3 to trivially forward all calls to these operation definitions.

After implementing all operations formally defined for operational reactive data-flow graphs in Section 3.3, the Scala implementation is ready to use for static analysis and transformation as we will demonstrate in Section 6.2. What remains is however is the implementation of reification and unreification to create a link to reactive frameworks and enable dynamic metaprogramming.

5.3 Reification of Meta Representation Graphs in Scala

We have already outlined general reification concerns like reification scheduling and the handling of partially reified reactive graphs in Section 4.2. These allow a straight-forward implementation of the shown concepts to reify the previously introduced meta representation structures implemented in Scala into reactive frameworks.

A major design goal for reification was to prevent coupling between the meta representation and REScala to retain the independence from actual frameworks that was established for the formal definitions in Chapter 4. For this reason, we have created a generalized *reifier* trait that the graph's nodes are depending on and that defines the reification program flow. The connection to REScala is only established by a concrete implementation of this interface through a REScala reifier.

5.3.1 The Generalized Reification and Unreification Process in Scala

Fig. 5.8 shows the abstract interface that is implemented by all reifiers for reactive frameworks. The publicly visible core methods are the ones for reifying each type of reactive node, as well as unreifying an existing node. Similar to the data-flow graph implementation, we also chose a mutable design for the reifier that allows it to store the necessary caching and state information needed to achieve a consistent connection between the graph nodes and the reified elements. The consequences of this can be seen in the public interface, as no separate methods for re-reification based on an existing state are required, and no saved state is returned when unreifying a node. Instead, this information is saved and restored implicitly by the reifier, which allows a high degree of freedom for the reifier's implementation.

```
trait  Reifier  {
  def reifyEvent[T](eventNode: EventNode[T]) : Event[T]
  def  reifySignal [A](signalNode: SignalNode[A]) : Signal[A]
  def  reifyEvt [T](evtNode: EvtEventNode[T]) : Evt[T]
  def  reifyVar [A](varNode: VarSignalNode[A]) : Var[A]
  def unreify(node :  DataFlowNode[_]): Unit

  protected[meta] def evaluateNecessaryReification(graph: DataFlowGraph): Unit

  protected[meta] def doReifyEvent[T](eventNode: EventNode[T]) : Event[T]
  protected[meta] def doReifySignal[A](signalPointer:  SignalNode[A]) : Signal[A]
  protected[meta] def createEvt[T]()  :  Evt[T]
  protected[meta] def createVar[A]()  :  Var[A]
}
```

Figure 5.8: Abstract interface for reifiers that implement the reification process for data-flow nodes.

Also visible in Fig. 5.8 is the internal interface for reifiers that is used by the meta representation's nodes during the reification process. The method *evaluateNecessaryReification* is a generalized version of the *reify-if-necessary* function that was shown in Def. 4.5, and is used to determine if firing an event may trigger potential observers. Instead of only performing an analyzing this based on a single fired origin node, it analyzes the whole graph at once. This allows more complex situations where multiple events may be fired at once or, as possible in REScala, signals also have observers, which require an immediate reification and behave differently than event observers shown in Chapter 4.

The remaining methods are used internally to actually perform the reification after necessary dependency checks have been performed by the publicly available ones. While *doReifyEvent* and *doReifySignal* check the reifier's cache and create elements based on the existing meta representations, *createEvt* and *createVar* are used within them as generators for fresh root nodes without any other dependencies.

While the shown interface does not necessarily require the actual reification implementation to follow a fixed order of processing steps, it was created with a general process in mind that is illustrated in Fig. 5.9. Calling any of the public reification methods first causes the cache being checked to avoid multiple reifications of the same node. If the node already has a reification, it is returned immediately without further actions required. Otherwise, the set of required reifications to satisfy the node's dependencies is determined in a way as formally defined in Def. 4.2. In a second preparation step, all relevant, not yet reified log entries of the graph are then gathered in their original order. As long as unreified entries remain, they are handled according to their logged operation type:

- The reification of node creations uses the already outlined internal interface methods to perform the necessary node creation within the reactive framework. After reification took place, the result is added to the cache. Additionally, if an existing state for the node exists that was saved from a previous reification, it is re-applied to the node. Creating a node and restoring its state may also take place in a single step depending on the used framework's interfaces and design.

- Node operations like firing an event or setting a reactive variable can only occur in the log after the according node has already been created. To retain the original operation order, the node is fired immediately so that only dependencies that were already existent when the log entry was added are affected.

- Adding an observer works similar to firing and setting nodes by forwarding the operation to a corresponding call on the reified node. For observers however it is also necessary to store themselves in the cache to allow a later handling of them during unreification.

After a log entry has been handled, it is marked as reified to avoid a duplicate reification of the same operation. Consequently, it is also removed from the queue of unhandled, relevant nodes for the current reification. The queue continues to be reified until no unhandled log entries remain. Since the originally requested node is always part of the scheduled reifications through the log, it must have been processed by finishing the scheduled reification of the log. Consequently, a simple lookup of the originally requested node in the cache is sufficient to get its reification.

Unreification takes place similarly as reification, is however slightly simpler than the reification process. In Fig. 5.10, the unreification process is outlined. As for reification, it is first checked if the whole unreification process can be ignored as the node was already unreifiedor has never been reified before. It unreification cannot be skipped, the set of nodes to unreify is determined, which can be achieved through the same function as for reification as we proposed in Section 4.2.

As no natural ordering of nodes is available for unreification to ensure a scheduling of nodes before their dependencies, it must be created manually to determine the unreification schedule. This can be done by a depth-first search of the nodes that returns them in post-order.

After preparing unreification, the actual process executed for each node is to first delete all existing observers and mark them as unreified in the log. Observers themselves don't contain any internal state and therefore can simply be re-created when a later re-reification is applied. The node itself on the other hand may contain state information that needs to be extracted and saved in collaboration with the used reactive framework. Finally, the node can be deleted from the cache and marked as unreified in the log.

Applying this procedure to all nodes computed for unreification ultimately restores a state that is consistent and safe for a later re-reification. Reified nodes themselves do not need to be manually

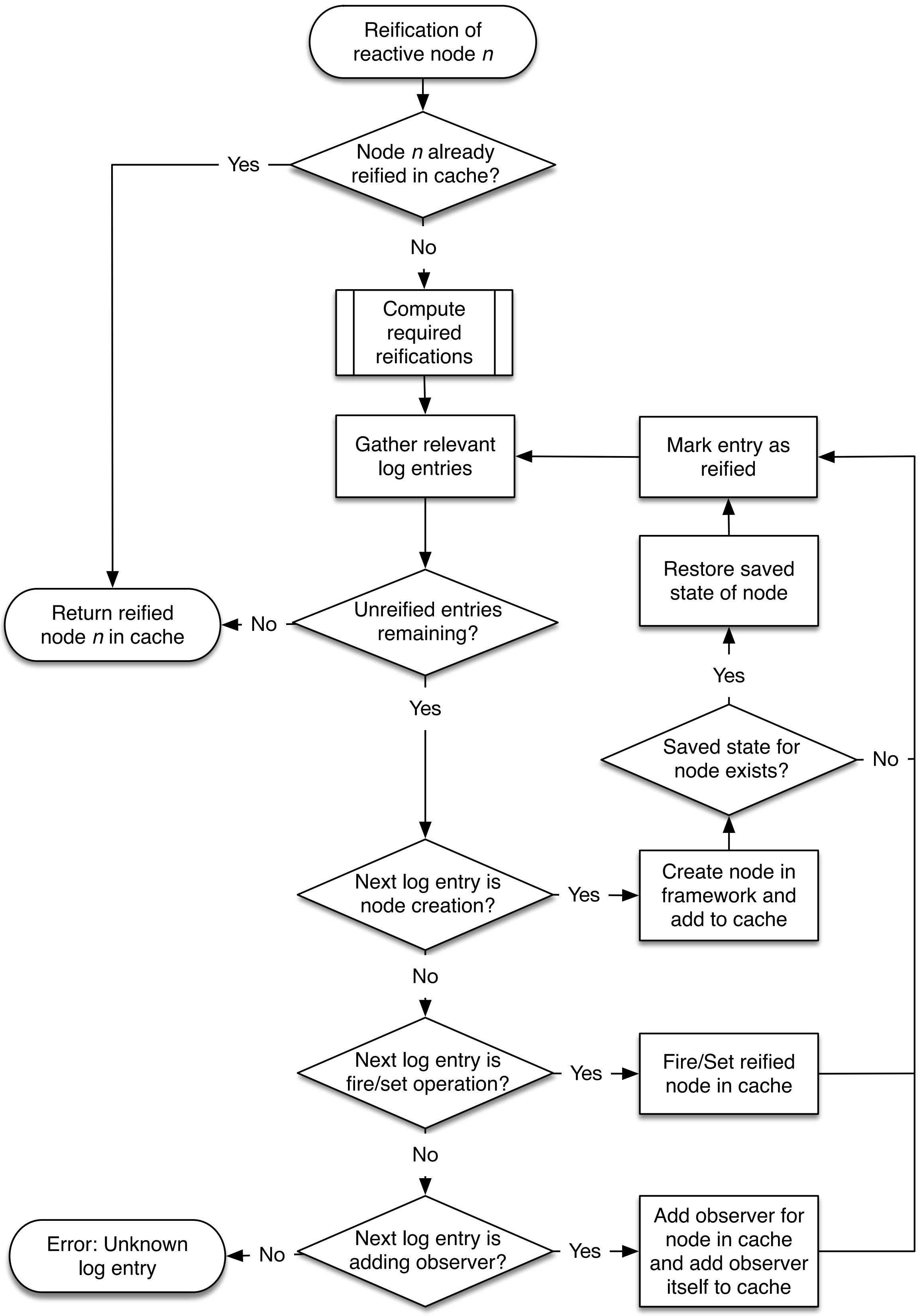

Reification of reactive node *n*
Node *n* already reified in cache?
Yes
No
Compute required reifications
Gather relevant log entries
Mark entry as reified
Restore saved state of node
Return reified node *n* in cache
No
Unreified entries remaining?
Yes
Yes
Saved state for node exists?
No
Next log entry is node creation?
Yes
Create node in framework and add to cache
No
Next log entry is fire/set operation?
Yes
Fire/Set reified node in cache
No
Error: Unknown log entry
No
Next log entry is adding observer?
Yes
Add observer for node in cache and add observer itself to cache

Figure 5.9: Flow diagram for the reification process.

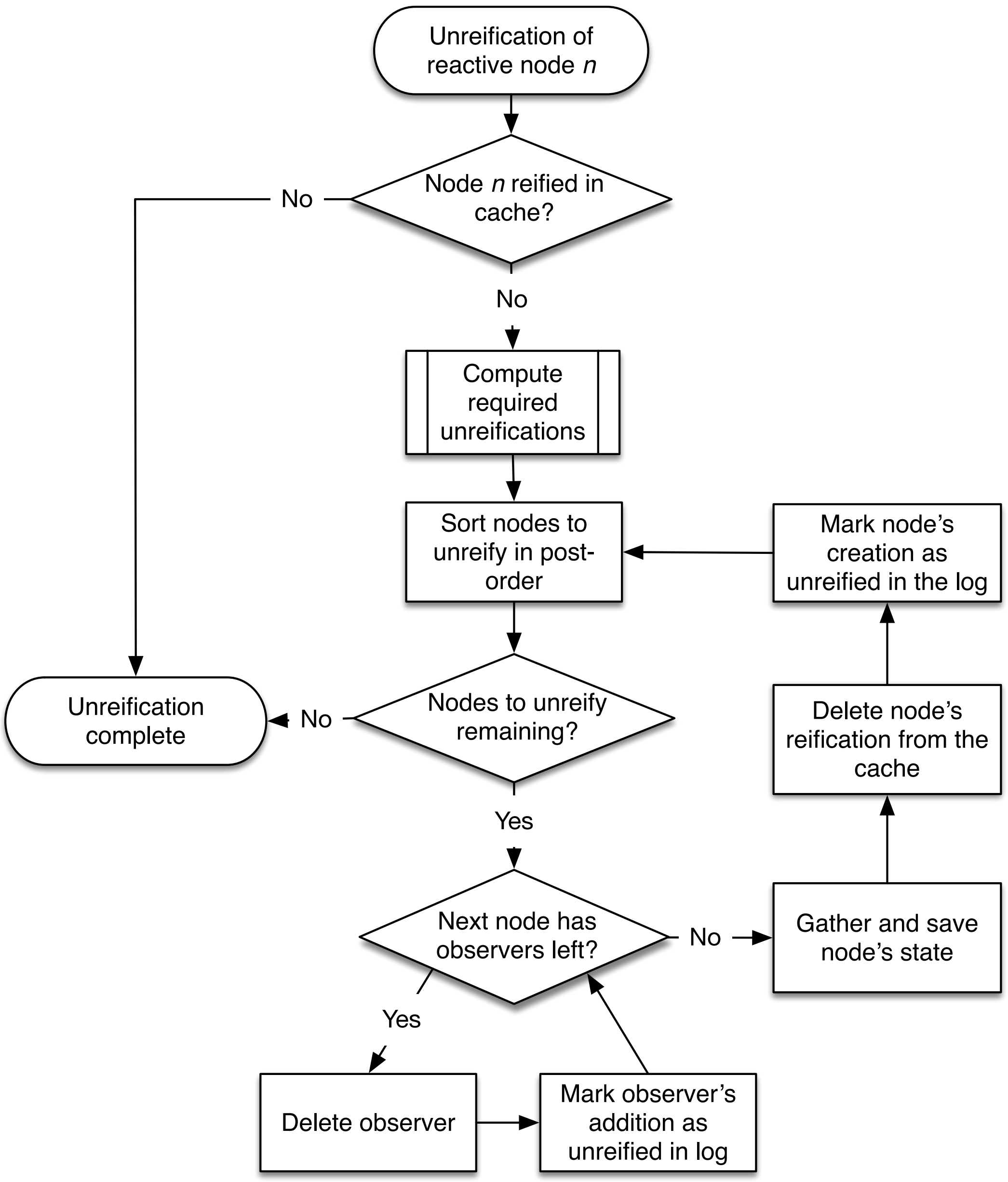

Unreification of reactive node *n*
Node *n* reified in cache?
No
No
Compute required unreifications
Sort nodes to unreify in post-order
Mark node's creation as unreified in the log
Unreification complete
No
Nodes to unreify remaining?
Delete node's reification from the cache
Yes
Next node has observers left?
No
Gather and save node's state
Yes
Delete observer
Mark observer's addition as unreified in log

Figure 5.10: Flow diagram for the unreification process.

deleted as the removal of all their observers ensures that even further operations may no longer result in any externally noticable effects. If available in the reactive framework, they may however even be disconnected from the remaining graph to avoid unnecessary computations and allow memory to be freed through mechanisms as garbage collection.

5.3.2 Implementation of REScala Reification and Unreification

In our implementation of the previously described reification process for REScala, we focussed on avoiding internal modifications of the framework. As our original goal was to create an external representation that is not depending on the internal framework design, reification should also be able to be applied from a mostly external viewpoint that uses only the interface that is already available for programmersusing REScala. As we will explain in this section, we have achieved this goal with one exception that required an extension of the original REScala code.

As we have outlined in Section 2.3, REScala contains several features that go beyond the simple management and propagation of reactive values. Especially the support for a wide range of different implementations of concurrency through propagation engines makes up such a strong part of REScala that it cannot be simply ignored when reifying a reactive graph. When creating or using reactive elements in REScala, it is necessary to also select an appropriate propagation engine and in some cases also a concorrency context to schedule a correct internal scheduling of operations.

Since REScala uses implicit parameters to make large parts of the concurrency interface transparent for the user, it is for most parts sufficient to adapt this design into the reifier's own implementation, so that implicit values are simply forwarded to REScala as they would be when directly working with its original interface. Further, we have decided to adopt this design to also make the reifier itself implicit when reifying or unreifying a meta representation node.

```
 1  import rescala.engines.CommonEngines.synchron
 2  implicit val reifier = new rescala.meta.EngineReifier
 3  val metaApi = new Api.metaApi(new ReactiveGraph())
 4
 5  val v = api.Var(1)
 6  val e = api.changed(v)
 7  var fired = 0
 8  api.set(v, 1)
 9  api.observe(e, ((x : Int) => { fired += 1 }))
10  api.set(v, 2)
```

Figure 5.11: Sample of the data-flow graph interface and the implicit reification to REScala elements.

A simple program using the meta representation as well as the implicit reification to the REScala framework can be seen in Fig. 5.11. The first two lines define the implicitlyused context for reification by setting the used REScala engine to a synchronous, non-concurrent implementation and the used reifier to the one described here that uses REScala as framework. The third line then sets up a fresh graph of the meta representation that can be accessed through the defined API. The next lines are mostly self-explanatory as they simply use the API to createreactive elements in the graph. Through the unified interface, the actual data-flow meta representation that is generated by the API methods becomes almost indistinguishable from the actual framework.

The defined reifier is actually used implicitly in line 8, as this is the first line that may trigger an actual reification in case there are already observers registered for the set variable. Since this is not the case, the first actual reification is triggered in line 10, that makes the already presented reification process trigger the reification of the whole graph as it exists at this point. The registered observer is reified as well, and it is immediately fired, which causes a side-effect that is not handled by the meta representation and changes the value of the variable *fired*.

To achieve the shown behavior, all the reifier has to add to the reification process as presented in Fig. 5.9 is the actual instantiation of REScala reactive elements. This can be performed by simply calling the appropriate methods for creating a new *Var* or *Evt* in the used REScala engine, or the appropriate interface methods in the REScala API. Unreification, while not implemented as an implicitly triggered process here, can also be achieved by a straight-forward implementation of the Behavior shown in Fig. 5.10. Additionally to deleting a reification from the cache, REScala also provides a feature to actually disconnect existing nodes from their dependencies, which is done by the reifier to make sure they can be handled by the garbage collector of the JVM.

The only feature that is not natively supported by REScala is the saving and restoring of a node's state. Adding this feature is unfortunately not easily possible since stateful nodes such as *Toggle* are internally converted into folding operations that emulate the desired behavior and only maintain an internal variable containing the current state information. There is no interface for extracting this internal variable and it is consequently not simply possible to extract this state without replicating the internal behavior of REScala of splitting nodes and therefore losing significant semantic information when unreifying the graph.

To solve this issue of saving and restoring state, there exist two possible workaround options:

1. Emulate the calculation state information within the meta representation, even for reified nodes, for example by keeping track of the amount and values it was fired with. Then re-apply this state on re-reification.

2. Extend the framework to support such a tracking internally and allow the saving and restoring of this record as a state value that can be treated as a black box by the meta representation.

Only the first option maintains the design goal of not modifying the used framework's internal implementation, and it is also the only one that allows a conversion of existing state of a single node between different reactive frameworks. it has several significant disadvantages. First, it distracts from the original idea of the meta representation to be separated from the actual values stored in the reactive graph and not emulate the propagation process or computations performed by the reactive framework. Second. it causes a significant overhead as the meta representation needs to replay the whole history of a node on each re-reification and has not enough information available to optimize redundant or otherwise irrelevant operations. Consequently we decided for the second option in our implementation.

To extend the framework, we have added an optional internal tracking to REScala that creates a log of all previous values of a signal, that is updated whenever an internal reevaluation occurs. Using an additional interface, it is possible to read this log and save it when unreifying a node in the reifier. Similarly, it is possible to restore the log and therefore the current value of a signal during signalinstantiation.

As already mentioned, the log can be treated as a black box within the reifier, which however also means that no type safety can be guaranteed for this interface. For this reason, and because the actual overhead in used memory resulting from the internal log is not covered in our evaluation in Chapter 6, the feature should be considered experimental and replaced once an actual extended state management is implemented for REScala.

6 Evaluation

The implementation of the reactive data-flow graph in Scala and its connection to REScala through reification and unreification shown in the previous chapter demonstrates that the shown meta representation concepts not only build a foundation for a theoretical model of reactive data-flow, but can also be practically implemented to work in connection to an existing reactive framework.

What determines the usability of the model in real-world applications are however its impact on performance and most importantly its suitability for graph analysis and transformation that were the motivation to define an external meta representation in the first place. In this chapter, we evaluate these properties through both *benchmarking* of graph construction and reification as well as *case studies* that exemplify situations where having a meta representation enables new functionality to be implemented.

6.1 Benchmarks

In the previous chapters, we have shown data structures for creating a data-flow meta-representation for reactive programs, as well as algorithms to transform this representation into executable reactive programs. We consider the implementation effort for both the construction of data-flow graphs and the reification process as reasonable, and we have demonstrated its feasibility by giving a sample implementation that interacts with the REScala framework. However, another factor that is of importance for users working with the demonstrated meta-representation is the resulting overhead on *run-time performance*.

To measure the performance impact of our meta-representation, we have conducted benchmarks to compare its run-time in comparison to pure REScala code. As scenario, we have chosen a graph in our meta-representation that originates in an event source and onto which we apply a variable number of consecutively chained mappings. For this graph, we measure two different values of run-time performance: First, we measure the time required to construct construct the program in both the meta-representation and the REScala framework. Then, we consider the throughput of fired event propagations in an already constructed graph.

To further analyze the impact of certain factors on the performance of both the pure reactive implementation and our meta-representation, we have selected two additional variables for both benchmarks: First, we consider the size of the reactive program, which is determined by the number of consecutive mappings applied to the original event. Second, we differentiate if a reification of the meta-representation is triggered in the scenario.

For all our measurements, we used the *Java Microbenchmark Harness* [18], which is already deployed in a ready-to-use state with the REScala framework. We did not use specific hardware to run the tests as we were only interested in relative performance values and not the exact runtimes. To avoid artifacts through JVM start-up or initialization, for each test we ran five warmup iterations and five iterations for our measurements, each counting the number of complete executions of our test program per millisecond over one second. The calculated 99.9% confidence interval for our measurements lies within a 15% range around our measured average for all measurements. We selected the non-concurrent synchronous engine of REScala for all our benchmarks as we did not support any concurrency with our own implementation and therefore only want to consider single-threaded performance.

The resulting measurements for the first benchmarked scenario are shown in Fig. 6.1. The chart is split into four different sizes of benchmarked data-flow chains. The grey bars represent the run-time of pure code of the REScala framework, while the blue ones visualize the pure construction run-time of the data-flow graph meta-representation. The yellow bars show the run-time required to both construct a meta-representation graph and reify it into REScala code.

For all results, the necessary run-time is expectably closely related to the size of graph and grows approximately linearly with its size. More of a surprise might be that the construction of the meta-

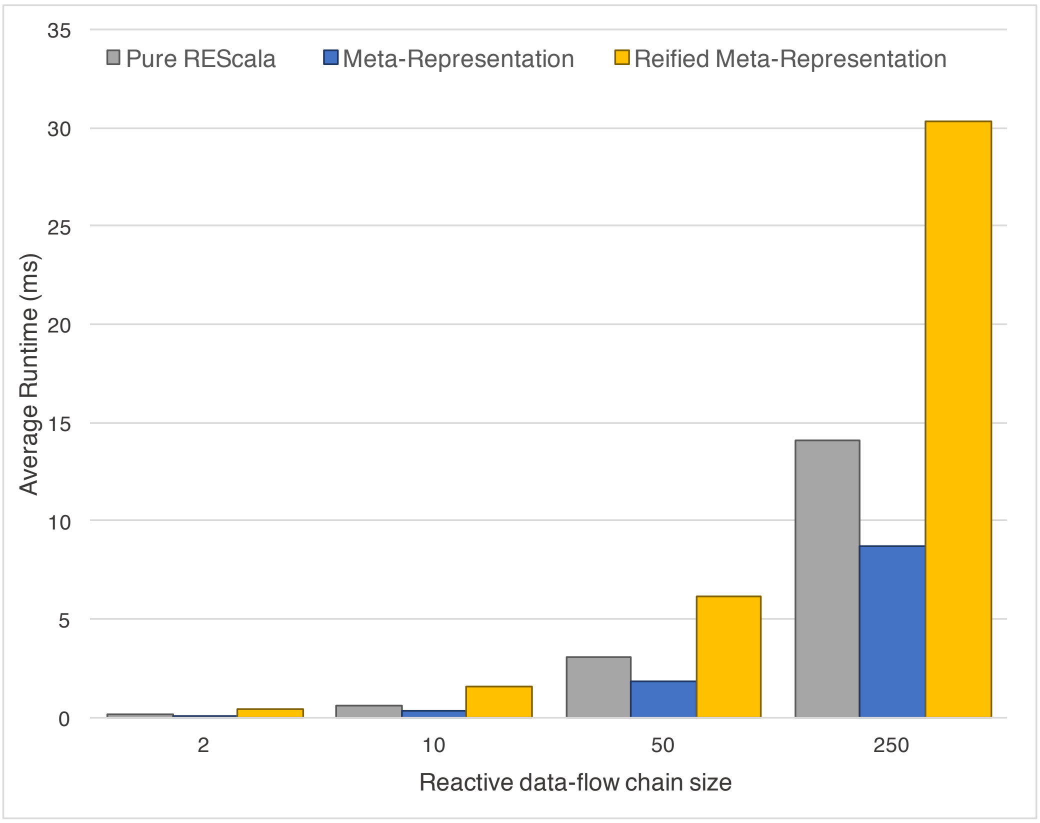


Figure 6.1: Measured run-time for construction of pure REScala programs and when using the meta-representation (lower = better).

representation graphs is significantly more cost-efficient than the creation of REScala programs. Our implementation needs to retain more semantic information about the graph layout than an internal representation like the one used in REScala requires. However, the set-up of structures and information required for the propagation of data-flow within REScala may cause this significant performance difference.

When also considering reification within our benchmarks, as this would be a prerequisite for executing propagation on the reactive graph, results change significantly: The reification appears to add an approximately constant factor of overhead to the pure construction of the graph that results in a run-time significantly larger than both constructions.

One reason for this is the fact that reifying the data-flow graph also includes constructing the resulting REScala graph. However, the measured run-time is also larger than the sum of both constructions. A possible reason for the caused overhead is the requirement to compute the graph dependencies that actually require reification. In our measurements, we triggered the reification of the last event node in the chain, which means that a recursive analysis of all incoming nodes is necessary to determine those that need to be reified. This process causes a significant computational overhead.

Our second benchmark evaluated the performance of update propagation in an already constructed graph. Fig. 6.2 shows the results of these measurements. As in our previous chart, we have executed the benchmarks for four different graph sizes. This time however we compare pure REScala code with a graph created using our meta-representation that has already been reified before our benchmark is executed. Additionally, we also performed measurements for a graph that is still unreified and does not contain any event observers. Due to the lazy reification semantics discussed in Section 4.4, no value propagation has to be performed in this case as the fired events can be ignored.

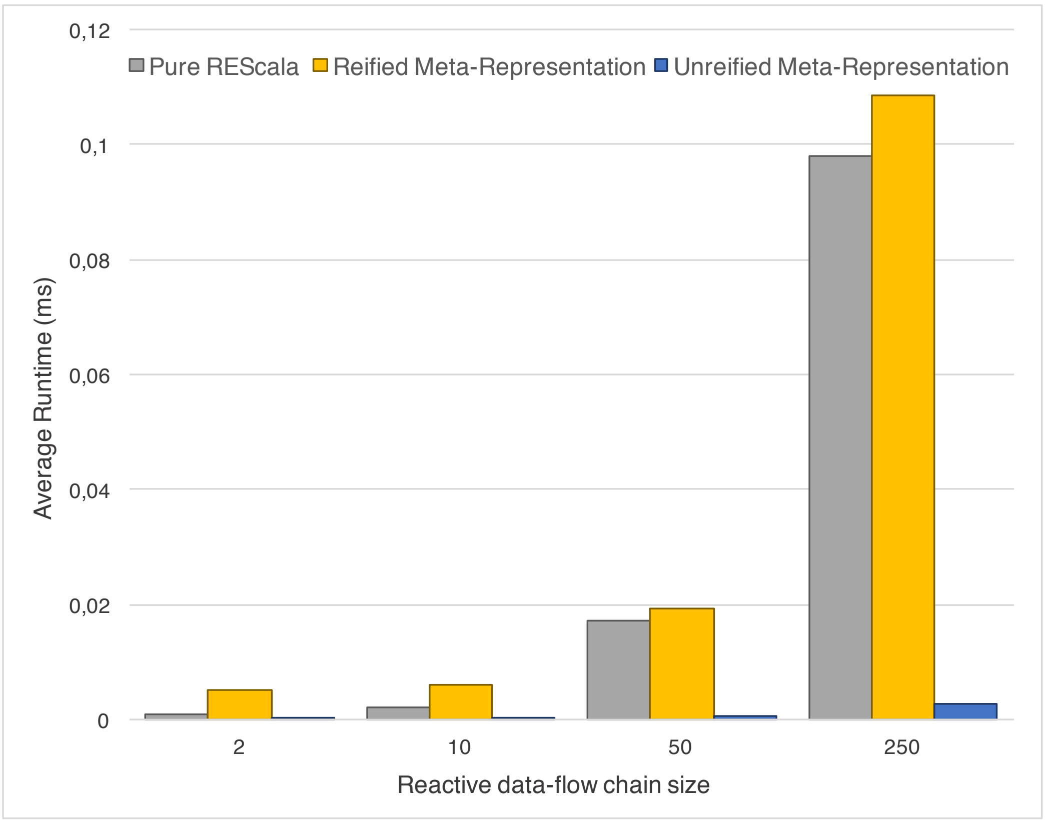


Figure 6.2: Measured run-time for firing a single event in pure REScala code and when using the meta-representation (lower = better).

When comparing the measured values from Fig. 6.2 to the ones from 6.1, our first observation is that the time required for propagating values is significantly lower than the time used for dependency construction. While construction times varied in an order of milliseconds, the bars in Fig. 6.2 show that asingle fired event can be propagated throughout our whole reactive sample program in a fraction of a millisecond, even for our largest tested graphs.

The general tendency of larger graph sizes requiring more computation time can be also observed in our second benchmark. However, when comparing pure REScala code with our reified meta-representation, the measured overhead varies significantly depending on the tested graph size. Firing a reified node in the data-flow graph triggers the REScala propagation process for underlying reified node and is therefore limited by the performance achieved by REScala. But while REScala shows significant performance increases even for very small graph sizes, our meta-representation seems to be limited by a hard performance cap that is not size-dependent.

This limit is most likely once again related the semantics of reification, as even for reified nodes, updates are first added to the graph log, the cache of reified nodes is checked and then the actual propagation is triggered. While this process becomes a less significant overhead for larger graphs as the required run-time barely increases with the graph size, it creates a strict constant performance limit for smaller graphs.

Also remarkable is the performance of the unreified meta-representation graph which achieves a drastically better performance than any other propagation. The reason for this is that for unreified graphs, no propagation process is necessary at all. As we are using scheme there that shows a behavior closely related to that of lazy evaluation, the fired event can simply be stored for later propagation during the reification process. In this specific case, it may even be discarded entirely as no lasting changes could have been triggered by it.

In Section 4.4 when we presented the semantics of implicit reification and unreification as they are used in our benchmarked implementation, we stated that it is an open question to determine an optimal strategy of targeted unreification of previously reified graph elements. Looking at the results obtained in our benchmarks confirms this as an important factor for creating a performance-efficient meta-representation. The impact of the reification process itself is signicant, while the overhead of propagating updates on the reified graph becomes negligible for larger graphs. Non-reified graphs can however save important computation time if the propagated values are irrelevant and their computation can be skipped. Yet, as the time scale for construction operations is much larger than that for propagation, these savings may rarely outweigh the required time for unreification and re-reification.

Even when not considering such implementation decision, the benchmarking results shown in this section also allowed us to analyze how the processes of creating, reifying and re-evaluating the meta-representation of a reactive program affect the resulting run-time performance in general. The actual performance in comparison to a pure implementation in REScala is highly dependent on the program's structure and the required amount of implicit reification. We havehowever demonstrated that the overall performance impact is not harming the feasibility of implementing even large-scale reactive data-flow chains.

6.2 Case Studies

The graph meta representation can be used for both purely non-destructive analysis operations as well as transformations that provide a benefit for programmers using a reactive framework. To show these capabilities, we have chosen two case studies demonstrating such scenarios: In Section 6.2.1 we show how the meta representation can enable optimizations of the data-flow structure created by a reactive program. Section 6.2.2 then shows how the representation can benefit the implementation of distributed reactive programs by allowing data-flow analysis and providing additional abstraction over a node's location for the user.

6.2.1 Data-flow Optimization

Achieving a high run-time performance is a common top-priority goal in programming. As mentioned in Section 2.2, the management of data-flow in reactive frameworks causes overhead and is therefore harmful when trying to maximize a program's performance. Modelling data-flow as a graph as we have presented in Chapter 3 however allows several possible optimizations that have been extensively researched in the domain of streaming [19, 20], compilation [21, 22] and hardware circuit design [23].

While optimizations of the data-flow may also be performed by reactive frameworks itself, their internal representation may not be sufficient for the necessary analysis and transformation steps, and adding the necessary capabilities may complicate the existing design and even increase the existing performance overhead. As an alternative, we propose the application of data-flow optimizations on the presented external representation, which can then be reified as shown in Chapter 4.

When running a reactive program, there are two sources of potential overhead: The necessary or potentially unnecessary recomputation of reactive elements themselves and the management of propagation by determining a glitch-free order of recomputation. To minimize these overheads, we propose the reduction of either the number of nodes or the number of edges in the reactive data-flow graph. Using the survey of data-flow stream optimizations conducted by Hirzel et al. [19], we chose two sample optimizations that are easy to implement and result in easily predictable benefits that are independent of the actual implementation:

1. *Fusion* merges a linear sequence of reactive elements into a single element that behaves semantically equivalent to the last element in the sequence. Linear and therefore fusible reactive operations are sequences of *mapping* and *filtering* of signals or events, where the used functions can easily be

applied sequentially and treated as a single operation without additional need for propagation or dependency computation.

A challenge when determining fusible elements is the consideration of external references. If a user-accessible reference exists for one of the inner fused nodes, it cannot be guaranteed that no additional dependencies are added or value queries are executed at a later point in time. The simplest approach is to only fuse nodes where no external references to inner nodes exist. It is however also possible to retain such references by duplication of the referenced node. While this may not result in a further decrease of the size of the optimized graph, unused duplicated nodes may be skipped during reification if their references are never used to add further dependencies. A sample of both fusion approaches applied to a data-flow graph with sequential mappings can be seen in Fig. 6.3.

2. *Redundancy elimination* replaces multiple reactive elements that share the same semantics by a single element. Consecutive application of this approach may even result in the step-wise elimination of subgraphs within a larger reactive program. Since reactive nodes that are not source elements are free of side-effects, semantic equivalency can be determined by a simple structural comparison of the graph. While different source elements are independent and therefore never semantically equivalent, all dependent reactive elements are semantically equivalent if all their dependencies are semantically equivalent and they apply the same operation.

   Fig. 6.4 shows an example where both branches of a data-flow graph apply equivalent operations on the same source variable. They can therefore be merged into a single linear graph by step-wise removal of the redundant nodes. References and dependencies onto any of the removed nodes need to be redirected to the retained equivalent node. In the shown example, this ultimately results in a *toggle*-node that is redundant itself as it only toggles between two references to the same signal node. Another optimization not shown in detail here which performs simple sanity-checks for situations like this one might remove the useless *toggle*-node in a later optimization step.

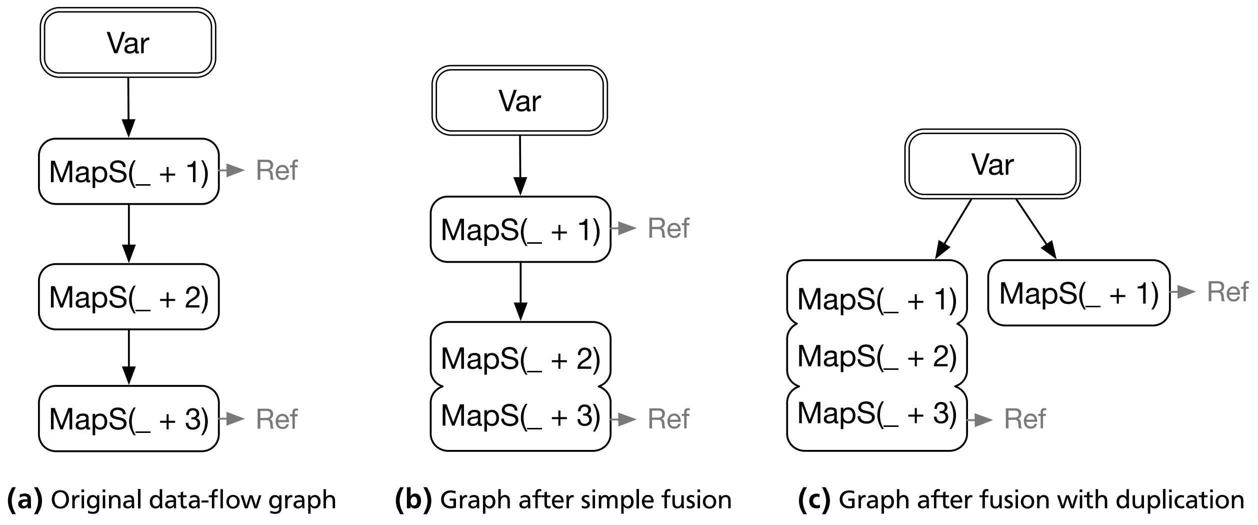
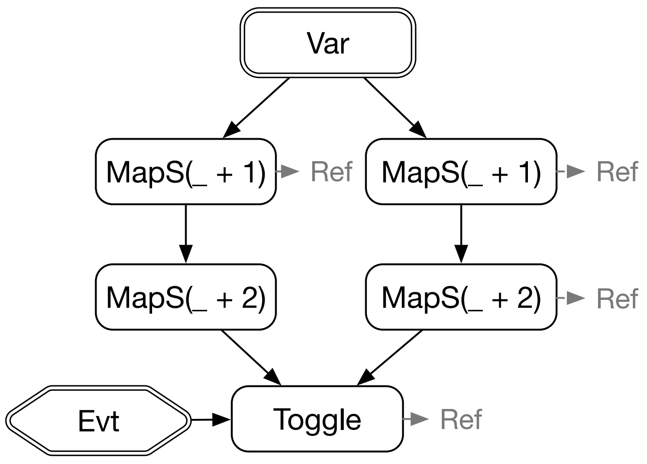
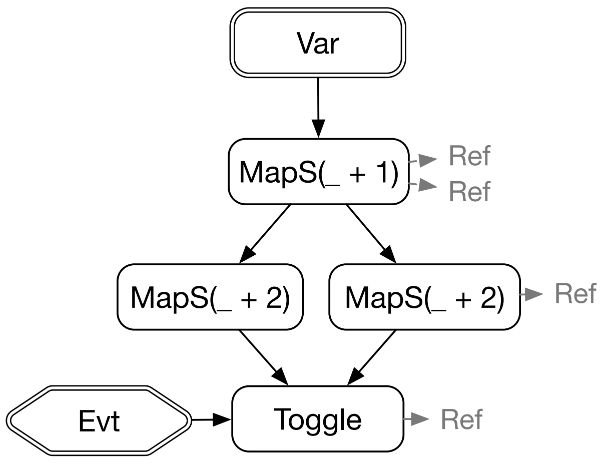


Figure 6.3: Fusion optimization applied to a data-flow graph with sequential mappings.

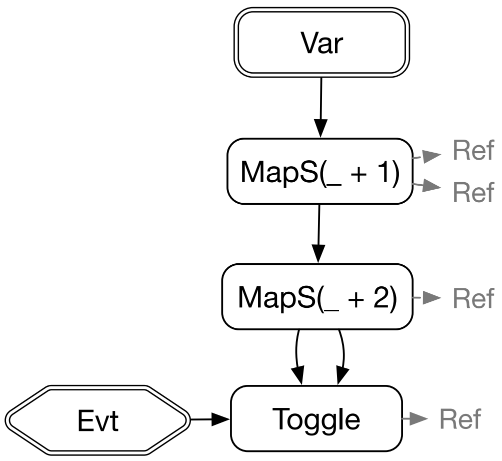
Both outlined optimizations can be applied to the semantic data-flow graphs as shown in Section 3.2 as well as the operational graphs as shown in Section 3.3. The availability of semantic information about each node and the traversal of the graph, similar to an AST, make the implementation of these and similar optimizations very efficient and straight-forward.



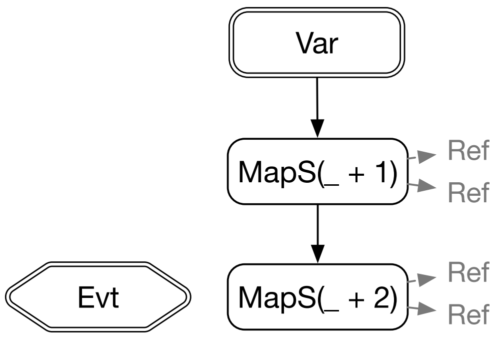
(a) Original data-flow graph



(b) Graph after first redundancy elimination



(c) Graph after second redundancy elimination



(d) Graph after additional simplification

Figure 6.4: Redundancy elimination applied to a data-flow graph with redundant branches.

For the operational model, it is however important to also consider the timing of node creation when fusing or replacing nodes. If one node was added before applying a firing or setting operation while another was added afterwards, they may not be considered semantically equivalent and also not be fused. When using an implicit reification approach as shown in Section 4.4, it is further necessary to ensure that the nodes and all affected dependencies are unreified before any optimization is performed to avoid discrepancies between the graph and its reification.

To demonstrate the practical applicability of the shown optimizations, we have implemented them in Scala based on the implementation shown in Chapter 5. Especially the additional level of indirection through mutable references introduced in Chapter 5 turned out to be very useful for implementing node replacement as performed during redundancy optimization.

Further optimizations and clean-up transformations like the mentioned check for redundant combinations of references to the same node are possible. Some useful optimizations like *operator reordering* however must be treated cautiously as they may alter the semantics of nodes like *fold* that have side-effects and therefore carry state information themselves. We will further discuss some of the optimizations as presented by Hirzel et al. that are not graph- but network-based in the next subsection.

6.2.2 Distributed Reactive Programming

Modern software development commonly involves the requirement to distribute software over more than one physical machine. This may be an inherent consequence of the software's use-case that could require a central server to manage data-flow between multiple client computes, or it may be simply the result of fulfilling a need of resources or processing power that is not available within a single computer. Efficiently handling the necessary communication between different processes over network while ensuring the intended program and data flow requires a careful software design that accounts for the resulting challenges [24].

An area of current research is to combine distributed computing concepts with reactive programming in the area of *Distributed Reactive Programming (DRP)* [1, 14]. When distributing reactive programs over a network of connected computers, well-known issues from distributed computing like unreliability, latency and highly dynamic network partitioning can affect core guarantees of reactive frameworks like glitch-freedom or the preservation of operation order.

The actual implementation of networking functionality and finding practical solutions for the arising issues are tasks that need to be solved by the actual reactive frameworks. Therefore, they do not fall into the high-level structural level that is covered by the graph representation presented in this thesis. To create efficient distributed programs and minimize the overhead resulting from costly network operations, it is however highly beneficial to analyze the data-flow within a program before and during its execution. We propose that the data-flow graph meta representation we have presented can be used to conduct analysis of this type and therefore provide a framework-independent insight into potentially beneficial networking layouts and optimizations.

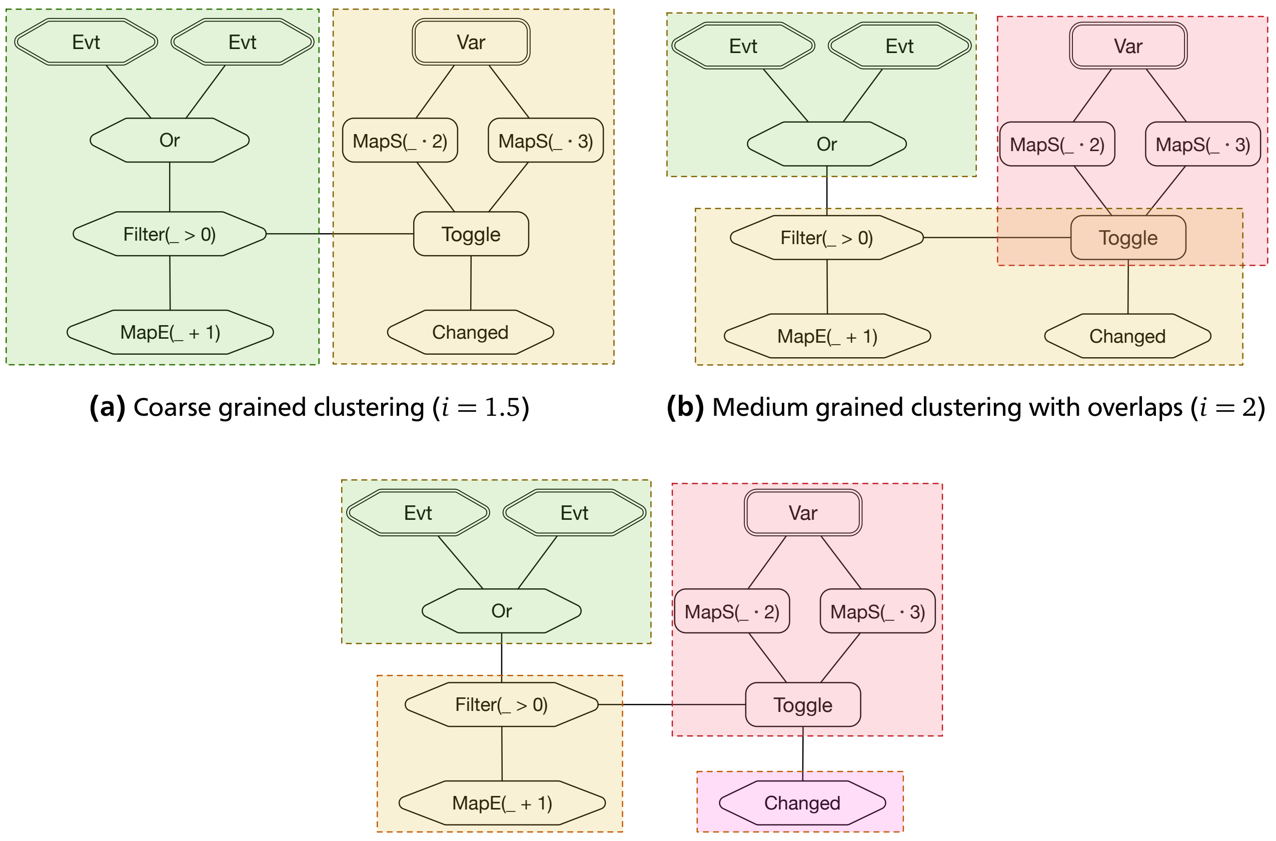
In distributed environments where the actual network layout can be chosen dynamically, it is beneficial to use *clustering* techniques that try to minimize the proportion of costly data-flow over the network to more efficient internal flow on a single machine. When a data-flow graph is available, this can be achieved by finding groups of nodes with a high degree of connectivity. Assuming a uniform distribution of data-flow in the graph, these nodes are more likely to exchange data with each other than other nodes. Therefore, it is beneficial to consider them as a cluster and move them to the same physical computer.

While a variety of different problem statements and approaches to graph clustering exist and extensive research has been conducted on their properties and relation [25], we have decided to apply the *Markov Clustering (MCL)* algorithm [26] to our data-flow graph representation. This algorithm uses a stochastic model to iteratively approximate node sets that are most likely to be connected by data-flow. It has a polynomial runtime, works effectively for many different graph layouts, and an open-source implementation in Java is available as part of the *Java Machine Learning Library* [27].

The MCL algorithm can be used in combination with any of the graph models we have presented in Chapter 3 as the minimum requirement to apply it is an adjacency matrix of the graph to be clustered. Since the average number of incoming data-flow edges for a single reactive node is usually not scaling with the graph size, reactive graphs tend to be sparse. Therefore, a map-based spare matrix representation can drastically reduce the required memory size compared to an array-like matrix storage.

While the MCL algorithm supports both undirected and directed graphs, it is recommended by its original authors to be used with undirected or mostly undirected graphs. This fits with our observation of its results being significantly better when ignoring the direction of data-flow edges during adjacency matrix generation. While the algorithm also supports weighted edges, we initially consider the same weight for all edges here.

Applying the MCL algorithm to the generated adjacency matrix causes it to iteratively increase the weight of edges between strongly connected nodes and then normalize the matrix. Through this process, inter-cluster edges are weighted down until they are abandoned by falling below a cut-off threshold. The resulting matrix only has the edges within each cluster remaining, which can then be extracted to convert the matrix back into sets of the clustered nodes.



(a) Coarse grained clustering ($i = 1.5$)

(b) Medium grained clustering with overlaps ($i = 2$)

(c) Fine grained clustering ($i = 3.2$)

Figure 6.5: Sample clusterings after applying the MCL algorithm with different inflation factors i.

Fig. 6.5 shows sample results of applying the MCL algorithm with different inflation parameters i to a small data-flow graph with ignored edge directions. The colored boxes indicate the clusters determined by the algorithm. The results indicate two drawbacks of this algorithm, the first one being the difficulty to control the number and size of the resulting clusterings. Unfortunately, it is not possible to directly relate the used inflation factor to the number of resulting clusters or their sizes, as these may vary drastically depending on the graph's layout. The second drawback of the MCL algorithm is that in rare cases it may produce overlapping clusters as in Fig. 6.5b. These can typically be avoided by selecting another inflation factor as in Fig. 6.5c, but still need to be considered when using the algorithm as part of an automatized toolchain.

Typically, an algorithm like MCL would require multiple runs with different inflation factors to find the desired clustering result for a given data-flow graph. The meta representation we have presented may help reduce the cost or amount of overhead necessary for each run as it allows a very straight-forward and potentially more efficient generation of a graph's adjacency matrix than internal representations used in reactive frameworks. This benefit becomes even more significant when considering reactive programs that are already in used within a distributed setting, where full layout information may not be directly available within a reactive framework itself.

The shown example of applying the MCL algorithm is just one example of an analysis that could allow or optimize distributed reactive environments. Even considering MCL, there is potential for further optimizations like using different initial edge weights for different types of nodes when using the algorithm on semantic or operational graphs. Another extension that may lead to significantly improved results is the measurement of actual flow data created during runtime as a more realistic edge weight. We will give a general outline about dynamic analyses that may be relevant for future research in Chapter 8.

7 Related Work

While no comparable attempts are known to us to create a meta representation specifically for reactive programs, the work presented in this thesis combines two branches that each have been the subject to diverse research in the past. The first one is the modelling of non-reactive data-flow for usage in program analysis and optimization. The second one is metaprogramming on a different semantic level that involves both meta representations and transformations of programs at compile-time and the dynamic access and modification of a program through itself as part of the *meta-object protocol*

7.1 Modeling of Data-Flow

The idea to model the data-flow of programs using graphs dates to the 1970s when they were applied to analyze programs written in the PL/I programming language [28]. Since then, they have been used in a wide array of areas, from networking [19, 20] to program analysis [11, 28], compilation and optimization [21, 29, 30] as well as hardware circuit design [23]. Due to the different requirements of each of these problem domains, specific specializations and extensions have been established for each of them.

Especially the task of analyzing data-flow in non-reactive programs bears resemblance to the models presented in this thesis. The fundamental difference between both tasks is however that in non-reactive graphs, data-flow from a node to another is bound to a specific point in time. This leads to typical analyses like required variable and object life-times [31] that are used as part of common compiler strategies and in register allocation. Such analyses don't are usually not applicable to reactive programs where signals and events can be updated implicitly by changes to their dependencies that are not determinable at compile-time.

An extension to the classic data-flow graph structure that is commonly used in compiler frameworks are *Control Data Flow Graphs (CDFG)*, which model flow of data and the control-flow of the represented program simultaneously [32]. They typically do this by using a two-level representation that models data-flow within basic blocks not interrupted by control-flow operations, and control-flow between these blocks that are then considered as atomic elements [28]. Since reactive programs typically don't have control-flow within the reactive flow network, this model is not applicable to reactive programs. An exception are however frameworks that support the nesting of higher-order program structures within signals like REScala. For these, a two-level representation of data-flow may be useful, which would however need to be reversed in comparison to existing models as it requires consideration of data-flow on a global and control-flow on a local level. We have not considered representing this type of higher-order embedding within our graph representation, which may be a topic of future research.

More closely related to reactive programming when representing actual data-flow structures are representations of data streams, for example in networks or when modelling data pipelines. As we have shown in Section 6.2, most use cases of modelling data-flow for streams, like optimization and placement of operators within a network are also applicable to reactive programs [19, 20]. Unfortunately, existing work in this area did not actually try to formally define the used models and usually assumes the applied data-flow structures to be manually set-up as a graph and then rolled out. Therefore, no meta-access to the resulting network of streams has been considered up to this point.

7.2 Metaprogramming

Two of the fundamental benefits of high-level programming languages over low-level machine code representation are the provided level of abstraction as well as a certain level of correctness validation resulting from a voluntary restriction of the programmer's capabilities. There are however situations when it can be desirable to get access to a program from an external view-point that considers the program code itself as viewable or even modifiable data. The common term for all variations of this type of programming is *metaprogramming*.

Metaprogramming can be divided on a general level into two subtypes: *Static metaprogramming* accesses program code before it is executed, for example during the compilation process or between compilation and execution. It can only rely on the code itself and may not make any assumptions about the runtime environment or user input. *Dynamic metaprogramming* is more powerful as it can access or even modify the program at run-time, and access concrete information about its state and environment. A subtype of dynamic metaprogramming is when a program gets access to itself by modifying its own structure or the semantics of its execution.

In this section, we will outline work in both of these areas and show their relation to the reactive meta representation we have presented in this thesis.

7.2.1 Static Metaprogramming

As all programs accessing code after or even while it is written up to the point when it starts being executed can be considered as static metaprogramming, a wide range of different research areas fall into this category. Even compilers themselves can be considered metaprograms, as they analyze and often also optimize program code while lowering it into another representation which is often assembly language or machine-independent bytecode.

In this section we will however focus on static metaprogramming in a more narrow definition that considers already finished program code and analyzes or transforms it in ways that do not result in a low-level form like it would be the case after compilation. Especially areas like recommender systems or refactorings could also be considered from a reactive perspective, but have too little in common with the type of meta representation presented in this thesis, which assumes an already functional reactive graph that can be analyzed.

The idea of enable access to a static representation of a program that is not simply code as plain text but a form that is already enriched with syntactic and semantic information have been established for decades [33, 34]. For program code, representation through an *Abstract Syntax Tree (AST)* that is often decorated with additional semantic information is wide-spread and has proven useful for for a wide array of usages. Two areas that highly benefit from this form of representation are hygienic macro expansion [35] and multi-staged compilation [36], which both use ASTs to partially lower parts of a program into a form that itself is still represented within the AST and therefore can be further analyzed or transformed.

Using ASTs as a meta representation for reactive programs is not a very applicable solution as they cannot connect the representation of reactive elements as objects or variables to the resulting permanent data-flow connections that are the core principle of reactive programming. These semantics are only established through the used reactive framework which is a dynamic process that cannot be reconstructed by pure analysis of the program's or the framework's AST.

The reactive meta representation we have presented uses a design inspired by that of ASTs, and can therefore share some of the same benefits. As we have demonstrated in this thesis, the representation of operations by different node types that each can carry parameters specific to their use is as useful in reactive data-flow graphs as it is in classical program transformations. Especially the ability to efficiently traverse the graph while applying directed transformations to specific nodes or node groups has proven useful in our case study about optimizations in Section 6.2.1.

Another representation of programs found in metaprogramming, especially in the context of optimization is Control Data Flow Graphs that we have already presented and compared to our representation in Section 7.1. The optimizations for reactive data-flow that we have presented in Section 6.2.1 are typically also executed using this representation when applied to non-reactive program data-flow. The same applies for static analysis, which uses similar techniques as optimizations and can be applied to both ASTs and a variety of data-flow graphs. We have only outlined some examples for static analysis in the context of distributed programming in Section 6.2.2, while other usage scenarios are thinkable.

7.2.2 Dynamic Metaprogramming

While static metaprogramming mostly varies in the time of application but in almost all cases targets preparing a program for execution, dynamic metaprogramming is applied with a wider array of goals. An example type of dynamic metaprogramming that is intuitively applied but almost all programmers in daily use is debugging, which allows to inspect the runtime behavior of programs. More sophisticated tools and methods exist that even allow modification to programs while they are executed, or even the way interpretation through a virtual machine or a runtime environment takes place.

Similar to static metaprogramming as outlined in the previous section, one of the main problems to solve for dynamic approaches is to find a well-suited representation of the program. A specific challenge is the connection of run-time information and static program structures like the AST. Before execution, programs usually are compiled to assembly language or at least a low-level representation that is optimized for efficient interpretation at the price of less easily available semantic information. This means that dynamic information like the current state of a partially executed program needs to be explicitly connected to the original, static program representation that carries high-level semantic information.

An approach to avoid the necessity of establishing such a connection between high-level and low-level semantics is *symbolic execution* [37]. In this analysis technique that bridges static and dynamic metaprogramming, a program is statically analyzed and assumptions about the dynamic behavior for specific input value ranges are made. A common application for this is formal verification of programs [38]. We could not identify existing research that attempts to apply such techniques specifically to reactive programs. We assume however that by analyzing reactive data-flow models like the one we have presented, similar attempts can be made that allow a better formal analysis of reactive programs.

For actual run-time analysis, relating the static and dynamic information about a program is considerably simplified when a program is directly interpreted instead of compiled. Consequently, functional programming languages like *Lisp* that are interpreted instead of compiled have been one of the earliest and thriving domains of dynamic metaprogramming. Ground-breaking work has been done with the definition of the *Meta Object Protocol (MOP)* for the *Common Lisp Object System* [39]. With this protocol, the semantics of object-oriented language constructs cannot only be represented and implemented in Lisp itself, but even be accessed, modified or re-defined through the programmer at run-time.

While having MOP-support would theoretically be a desirable property for all object-oriented languages as it enables a whole new level of flexibility, it comes with several restrictions and disadvantages [40]. For languages like LISP, having an MOP that also acts as a constant proxy for all internal interpretation of object-oriented code means a significant reduction of performance. Even more problematic is however the application to compiled languages, which would require the emulation of any object-oriented code execution by an interpreter executed at run-time, which removes almost all performance benefits of compilation. As very few programs require MOP capabilities and only programmers with a deep understanding of object-oriented semantics are able to utilize them, such an overhead is not acceptable for most real-world software

This dilemma when creating MOPs can also be identified when considering the data-flow meta representation in this paper: The ideal goal for reactive programming would be to have a meta representation like the presented one that allows a full meta-definition of the propagation process which can then be applied directly without even requiring any additional reactive framework. Such an implementation would allow a maximum of flexibility and enable the user to change the reactive semantics of his program on-the-fly and even during the propagation process. It would however also mean that a full emulation of the user-sided representation of the reactive graph needs to be employed for all procedures within the reactive data-flow processing.

The data-flow graph model we have shown employs a trade-off by mixing a high-level meta representation with a dynamic reification of its components into a lower-level representation through a reactive framework. This solution resembles the concepts of *reflection* and *just-in-time compilation (JIT)* found in dynamic metaprogramming for compiled but managed languages like Java [41, 42] and those based on the .NET framework [43].

While reflection is often used as an umbrella term for all features that allow programs self-inspection and self-modification including MOP, we will use the term explicitly to describe the ability to programs to access and modify their own structure but not to modify the fundamental semantics of their programming language. Compared with MOP, reflection is typically easier to realize as it does not require the programming language's semantics themselves to be accessible at run-time. Instead, it is only necessary to retain a mapping of the original, object-oriented structure after compilation that can then be modified on-the-fly. This is a feature that is typically already available in languages managed by a virtual machine to allow support for debugging or just-in-time compilation. Reflection is commonly employed in middleware like testing environments and has been used to efficiently implement programming paradigms like *aspect-oriented programming* [41, 44, 45].

The meta representation we have presented shares a conceptual similarity with reflection as it allows reactive programs to access and modify their own structure, while not giving direct access to the implementation of their reactive data-flow. This means a trade-off that lowers this degree of operational flexibility for a more efficient processing during run-time. Noticeably, the REScala framework used in our research already employs a certain degree of meta-access of its implementation through its support for interchangeable propagation engines that allow different implementation styles in the context of concurrency. Future research on lifting this support into an abstract form into the data-flow graph meta representation could result in more reflective access without sacrificing significant amounts of performance.

Like reflection, just-in-time compilation is another feature that is supported by modern programming language virtual machines [46] and even for interpreted languages [47]. It allows the analysis of code that would normally be interpreted and its optimization and compilation during run-time utilizing dynamic information. While the main target of this process is typically the optimization of run-time performance, the necessary analysis and transformation steps come at the price of a computational overhead that has a negative performance impact. Therefore, a benefit-cost ratio of the ideally non-recurring compilation process and the long-term performance increase when recurrently executing the optimized code needs to be considered.

A process similar to just-in-time compilation has been presented in Chapter 4 through reification and unreification of the reactive graph meta representation. In the presented model for reactive graphs, a similar benefit-cost analysis is necessary when considering the reification and unreification of reactive elements. While, in contrast to just-in-time compilation, at least a one-time lowering of nodes into a reified form is necessary, it is up to future evaluation to determine strategies to recurrently unreify and re-reify elements to apply optimizations onto them during run-time.

8 Conclusion and Future Work

In this thesis, we have presented a meta representation for reactive data-flow graphs that can be used as both a theoretical foundation for modelling reactive data-flow and to implement, analyze and transform reactive programs through metaprogramming. We have demonstrated how a reification into and from an actual reactive framework can be realized both explicitly and implicitly, and what impact an implicit reification has onto a program's propagation semantics. Further, we have outlined how to decorate the meta-representation graph with dynamic information gathered at run-time.

We have demonstrated the applicability of our models and evaluated them through an implementation in the Scala programming language that can be implicitly reified into a representation in the REScala reactive framework. Our benchmarks have shown that the performance impact of this process is measureable but not excessive, and may in certain cases even be beneficial. In our case studies, we have further demonstrated how our meta-representation enables dynamic optimization and data-flow analysis that may be particularly useful for the research area of distributed reactive programming.

Finally, we have shown relations between the presented approaches and other previous and ongoing research in metaprogramming. Based on these, we want to summarize possible directions of future work in reactive metaprogramming that can be founded on this thesis:

- While we have presented some examples of potential uses of a data-flow graph meta-representation, the provided list of examples was certainly not exhaustive. More transformations that are specific for certain use-cases of reactive programs may be found and tried, as well as analyses that affect other areas than distributed programming. Especially the field of code verification may be of additional research interest since having access to semantic data-flow information can enable new higher-level insights into program's run-time behavior. We also didn't consider any data-flow optimizations that may provide specific benefits for concurrent reactive programs, for example by facilitating pipelining.

- As we have illustrated in the discussion of our benchmarking results, it is difficult to determine the ideal time and quantity of implicit unreification to take place. While unreification can be necessary to perform transformations on the data-flow graph, it may also bring performance benefits to give up an existing reification to achieve a lazier propagation of values. Finding an optimized implicit unreification strategy is a task that is still open for research, especially when the shown concepts may be applied in performance-critical scenarios.

- Similarly, we also have not evaluated the efficiency of reification and unreification to other reactive frameworks than REScala. While we tried not to tailor our code to a specific framework, other reactive implementations can deviate from the underlying definitions we have used as the foundation for our design [1]. Especially a reification using frameworks a smaller feature-set than REScala, such as those supporting only either signals or events, may be challenging to fit with the presented models.

- The presented meta-representation considers each element in a reactive graph to have statically defined semantics and cannot explicitly represent higher-order control-flow within a single node, as it is enabled by certain reactive frameworks, including REScala. To allow the modelling of this control-flow embedded within the reactive graph, a hybrid representation similar to those used in compiler analyses and optimizations [28, 32] may be beneficial. This and similar extensions to our meta-representation could enable a bridge between the reactive semantics modelled by the data-flow graphs we have presented and regular program flow that they are merged into.

- A metaprogramming extension that potentially requires significantly more design and implementation effort but looks very promising is the addition of formalized propagation semantics to the data-flow graph model. This would increase the representation capabilities of the meta-representation onto the level of an autonomously working meta object protocol [39]. The result of this would be the ability to not only access and transform reactive data-flow but also every step of its propagation semantics dynamically from within the reactive program itself without any framework support or necessary reification process. This would enable new types of dynamic metaprograms as outlined in Section 7.2.2, and additionally, also benefit the general understanding of the fundamental semantics of data propagation in reactive graphs.

Bibliography

[1] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A survey on reactive programming. *ACM Computing Surveys (CSUR)*, 45(4):52, 2013.

[2] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming*, pages 159–187. Springer, 2003.

[3] 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*, pages 25–36. ACM, 2014.

[4] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[5] Ingo Maier, Tiark Rompf, and Martin Odersky. Deprecating the observer pattern. Technical report, 2010.

[6] Stefan Berner, Martin Glinz, and Stefan Joos. A classification of stereotypes for object-oriented modeling languages. In *International Conference on the Unified Modeling Language*, pages 249–264. Springer, 1999.

[7] Michiaki Tatsubori, Shigeru Chiba, Marc-Olivier Killijian, and Kozo Itano. Openjava: A class-based macro system for java. In *Workshop on Reflection and Software Engineering*, pages 117–133. Springer, 1999.

[8] Conal Elliott and Paul Hudak. Functional reactive animation. In *ACM SIGPLAN Notices*, volume 32, pages 263–273. ACM, 1997.

[9] Gregory H Cooper and Shriram Krishnamurthi. Frtime: Functional reactive programming in plt scheme. *Computer science technical report. Brown University. CS-03-20*, 2004.

[10] Antony Courtney. *Frappé: Functional Reactive Programming in Java*, pages 29–44. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[11] Alan L Davis and Robert M Keller. Data flow program graphs. 1982.

[12] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Weslay Professional, 4th edition edition, 2011.

[13] Martin Odersky. Scala language specification version 2.11, 2014. URL `http://www.scala-lang.org/files/archive/spec/2.11/`.

[14] 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*, pages 37–48. ACM, 2013.

[15] Joel Jones. Abstract syntax tree implementation idioms. In *Proceedings of the 10th conference on pattern languages of programs (plop2003)*, page 26, 2003.

[16] Peter John Brown. Writing interactive compilers and interpreters. *Wiley Series in Computing, Chichester: Wiley, 1979*, 1, 1979.

[17] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 273–280. ACM, 1989.

[18] Oracle. Java microbenchmarking harness. URL `http://openjdk.java.net/projects/code-tools/jmh/`.

[19] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4):46, 2014.

[20] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 49–49. IEEE, 2006.

[21] Etienne Morel and Claude Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, 1979.

[22] Robert Fitzgerald, Todd B Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: An optimizing compiler for java. *Software-Practice and Experience*, 30(3):199–232, 2000.

[23] Giovanni De Micheli. *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994.

[24] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems*. Prentice-Hall, 2007.

[25] Satu Elisa Schaeffer. Graph clustering. *Computer science review*, 1(1):27–64, 2007.

[26] Stijn Van Dongen. A cluster algorithm for graphs. *Report-Information systems*, (10):1–40, 2000.

[27] Thomas Abeel, Yves Van de Peer, and Yvan Saeys. Java-ml: A machine learning library. *Journal of Machine Learning Research*, 10(Apr):931–934, 2009.

[28] Frances E. Allen and John Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137, 1976.

[29] Jeffrey M Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9):724–736, 1978.

[30] Ken Kennedy. *A survey of data flow analysis techniques*. IBM Thomas J. Watson Research Division, 1979.

[31] Cristina Ruggieri and Thomas P Murtagh. Lifetime analysis of dynamically allocated objects. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–293. ACM, 1988.

[32] Said Amellal and Bozena Kaminska. Scheduling of a control data flow graph. In *Circuits and Systems, 1993., ISCAS'93, 1993 IEEE International Symposium on*, pages 1666–1669. IEEE, 1993.

[33] Robert D Cameron and M Robert Ito. Grammar-based definition of metaprogramming systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(1):20–54, 1984.

[34] Ole Lehrmann Madsen and C Nogaard. An object-oriented metaprogramming system. In *System Sciences, 1988. Vol. II. Software Track, Proceedings of the Twenty-First Annual Hawaii International Conference on*, volume 2, pages 406–415. IEEE, 1988.

[35] Eugene Kohlbecker, Daniel P Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 151–161. ACM, 1986.

[36] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *ACM SIGPLAN Notices*, volume 32, pages 203–217. ACM, 1997.

[37] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7): 385–394, 1976.

[38] Alberto Coen-Porisini, Giovanni Denaro, Carlo Ghezzi, and Mauro Pezzé. Using symbolic execution for verifying safety-critical systems. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 142–151. ACM, 2001.

[39] Gregor Kiczales, Jim Des Rivieres, and Daniel Gureasko Bobrow. *The art of the metaobject protocol*. MIT press, 1991.

[40] Gregor Kiczales, J Michael Ashley, Luis Rodriguez, Amin Vahdat, and Daniel G Bobrow. Metaobject protocols: Why we want them and what else they can do. *Object-Oriented Programming: The CLOS Perspective*, pages 101–118, 1993.

[41] Ira R Forman, Nate Forman, and John Vlissides Ibm. *Java reflection in action*. Citeseer, 2004.

[42] Thiemo Bucciarelli. Just-in-time compilation. *Institute for Software Engineering and Programming Languages*, 2016.

[43] Kevin Hazzard and Jason Bock. *Metaprogramming in. NET*. Manning Pub, 2013.

[44] Gregory T Sullivan. Aspect-oriented programming using reflection and metaobject protocols. *Communications of the ACM*, 44(10):95–97, 2001.

[45] Wolfgang Schult and Andreas Polze. Aspect-oriented programming with c# and. net. In *Object-Oriented Real-Time Distributed Computing, 2002.(ISORC 2002). Proceedings. Fifth IEEE International Symposium on*, pages 241–248. IEEE, 2002.

[46] Andreas Krall. Efficient javavm just-in-time compilation. In *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, pages 205–212. IEEE, 1998.

[47] Maxime Chevalier-Boisvert, Laurie Hendren, and Clark Verbrugge. Optimizing matlab through just-in-time specialization. In *International Conference on Compiler Construction*, pages 46–65. Springer, 2010.