Staging reactive programming

Staging von Reaktiver Programmierung
Bachelor-Thesis von Markus Hauck
Oktober 2013

Text

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Informatik
Software Engineering

Staging reactive programming
Staging von Reaktiver Programmierung

Vorgelegte Bachelor-Thesis von Markus Hauck

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

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

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

Darmstadt, den 4. Oktober 2013

(Markus Hauck)

Abstract

Reactive applications are a wide class of computing systems. Despite being studied for a long time, reactive applications are hard to write and maintain. Reactive programming (RP) is a recent paradigm that overcomes the drawbacks of the traditional Observer pattern. In the reactive paradigm the relations among reactive entities are defined in a declarative way and the language runtime automatically takes care of the updates of dependent values.

RP has proved an effective paradigm to design and implement reactive systems. However, two factors limit its applicability. On the one hand, current RP implementations have a huge performance impact compared to manual implementation of the reactive features. On the other hand, RP frameworks often require custom compilers and preprocessors that limit the applicability of the paradigm. Implementing RP as an embedded domain-specific language (DSL) solves this issues at the price of awkward syntactic burdens.

We implement OptiReactive, an embedded DSL for RP, that uses a recent approach to multistage programming, Lightweight Modular Staging, to perform runtime code transformations. Based on this technology, we develop several general purpose optimizations and syntactic transformations to improve the performance and enrich the syntax of RP. Noticeably, our solutions are general and applicable to other RP frameworks, while current RP research only exploits optimizations that are specific to the underlying implementation.

The evaluation shows that our optimizations improve the performance of our benchmarks and that the syntactic transformations do not have a negative impact on the runtime performance.

Zusammenfassung

Reaktive Anwendungen sind eine weit verbreitete Klasse von Computerprogrammen. Trotz langer Studien sind reaktive Programme schwer schreib- und wartbar. Reaktive Programmierung (RP) ist ein modernes Programmierparadigma, welches die Nachteile des traditionell verwendeten Beobachter-Musters (Observer pattern) beseitigt. In der Reaktiven Programmierung werden die Beziehungen zwischen reaktiven Elementen auf eine deklarative Art definiert und zur Laufzeit automatisch aktualisiert, unter Berücksichtigung der Abhängigkeiten.

Es hat sich gezeigt das RP ein sehr effektiver Programmierstil für das Entwerfen und Implementieren von reaktiven Systemen ist. Allerdings ist die Anwendbarkeit durch zwei Faktoren limitiert. Auf der einen Seite haben aktuelle Implementierungen von RP, verglichen mit dem manuellen Implementieren der entsprechenden reaktiven Features, einen sehr negativen Einfluss auf die Leistung. Auf der anderen Seite erfordern Frameworks für RP oft modifizierte Compiler und Präprozessoren, welche die Anwendbarkeit einschränken. Die Implementierung von RP in Form einer Domänenspezifischer Sprache löst diese Probleme, bringt aber meist syntaktische Lasten mit sich.

Wir implementieren OptiReactive, eine eingebettete domänenspezifische Sprache für RP, welche einen kürzlich präsentierten Ansatz für Multistage Programmierung, Lightweight Modular Staging, verwendet, mit welchem es möglich ist Programmcode Transformationen zur Laufzeit durchzuführen. Basierend auf dieser Technologie entwickeln wir mehrere Optimierungen und syntaktische Transformationen um die Performanz und Syntax von RP zu verbessern. Ein wesentlicher Aspekt ist, dass unsere Lösungen allgemein und daher anwendbar auf andere Frameworks für RP sind, während sich momentane Forschung auf diesem Gebiet auf Optimierungen beschränkt, welche von der konkreten Implementierung abhängig sind.

Die Evaluierung zeigt, dass unsere Optimierungen die Laufzeit unserer Beispielprogramme verbessern und die syntaktischen Transformierungen keinen negativen Einfluss auf die Ausführungszeit haben.

Contents

List of Figures

List of Listings

1 Introduction

Reactive programming has been proposed as a viable solution to implement reactive applications in a composable and less error-prone way. In this programming paradigm, reactive applications are defined in terms of time-changing values, and expressions that relate them. As a result, reactions are specified in a declarative way, avoiding the need for callbacks, whose drawbacks have been known in literature for a long time. Functional reactive programming was introduced in form of a composable library for animation [EH97]. The same concepts have since then been incarnated in various designs [NCP02, MGB+09]. More recently, there have been different implementations, fusing object-oriented and reactive programming in form of events or signals (behaviors) [MRO10, SHM14, GSM+11]. Many of the existing implementations feature performance optimizations, often with solutions that are specific to the underlying framework.

The concept of staging refers to the compilation steps of a computer program. In fact, every program written in a compiled language has at least two stages, *compilation* and *execution*. Between the two stages, the compiler may apply optimizations to improve the performance of the final program. Multistage programming introduces one or more intermediate stages, the programmer can leverage to apply transformations, including code optimizations.

This work aims to propose an optimization technique for reactive programming. We make use of a recently presented approach to multistage programming, called “Lightweight Modular Staging” [RO10] and the associated compiler framework Delite [Rom12]. Access to the intermediate stages provides us with the ability to do syntactic transformations and program optimizations.

The detailed contributions presented in this work are:

- We developed SimpleReactive, a library for reactive programming in Scala that implements the minimal concepts of reactive programming, required to demonstrate our approach.
- We propose syntactic transformations at the DSL level to ameliorate the usability of the SimpleReactive library.
- We propose optimizations that can be applied on SimpleReactive to improve the performance of reactive programs.
- We implement OptiReactive, a DSL for reactive programs built on top of SimpleReactive that implements the aforementioned transformations using Lightweight Modular Staging and Delite.

The rest of the thesis is structured as follows. Chapter 2 introduces the frameworks and techniques used to realize our implementation. Chapter 3 discusses the design and the implementation of SimpleReactive and OptiReactive. Chapter 4 presents syntactic transformations and optimizations together with their implementation in OptiReactive. Chapter 5 evaluates the performance of OptiReactive. Chapter 6 discusses related work, Chapter 7 concludes the thesis and outlines future work.

2 Technical Background

This chapter provides an overview of the basic concepts used in the rest of the thesis that are necessary to understand the design and implementation of OptiReactive. First we introduce multistage programming at a high level of abstraction, then we present the main technologies used in the rest of the thesis, Scala-Virtualized, Lightweight Modular Staging (LMS) and the Delite framework.

2.1 Multistage Programming

In case of a compiled language, executing a program consists of at least two stages: *compile-time* and *run-time*. The usage of a program generator such as Yacc or ANTLR introduces another stage, *program generation*. "A multi-stage program is one that involves the generation, compilation, and execution of code, all inside the same process" [TS00]. Staging can improve the performance of a program by applying changes or rewrites to its internal structure, before it is actually run.

2.2 Scala-Virtualized

Scala-Virtualized [MRHO12] is a modification of the Scala compiler and standard library. Its purpose is to provide enhanced support for embedded DSLs using the Scala programming language. Key features are the overloading of control structures like loops and pattern-matching, variable assignment and creation, as well as object instantiation. In addition to the overloadable constructs, it also allows to add new infix methods on existing types. Listing 2.1 shows an example on how to define a new infix method, +, on an existing type `Point`.

```
1  case class Point(x: Int, y: Int)
2
3  def infix_+(p1: Point, p2: Point): Point = {
4    (p1,p2) match {
5      case (Point(x1,y1),Point(x2,y2)) =>
6        Point(x1+x2,y1+y2)
7    }
8  }
9
10 Point(21,0) + Point(21,42) // => Point(42,42)
```

Listing 2.1: Extension methods

In line 3-8 the new extension method is defined. The "`infix_`" prefix is stripped from the method name, which enables the call in line 10. Technically, extension methods can also be defined using normal Scala by using an implicit conversion. Compared to Scala-Virtualized's extension methods, which are very lightweight, the alternative with implicits requires more boilerplate code, typically involving a class and a conversion method.

Control structures can be overloaded by overriding the specific method provided by Scala-Virtualized. One example is the `__ifThenElse` method, describing the standard conditional expression used in Scala. Programmers can then alter the semantics of all conditional expressions by overwriting the designated method.

As an example, listing 2.2 shows the definition of an alternative if-statement that prints an additional message, if the condition evaluates to true.

Scala-Virtualized plays a key role in "Lightweight Modular Staging", where it is used to make Scala constructs like condition expressions work with staged expressions and to define new infix methods on existing types.

2.3 Lightweight Modular Staging

Lightweight Modular Staging (LMS) is *"a library-based multistage programming approach that [...] uses only types to distinguish between binding times"* [RO10]. The framework uses Scala-Virtualized to allow the staging of built-in language features.

```scala
override def __ifThenElse[T](cond: => Boolean,
  thenp: => T, elsep: => T): T = cond match {
  case true =>
    println("Condition was true.")
    thenp
  case false =>
    elsep
}

if (true) println("True.") else println("False.")
// prints "Condition was true." and "True."
```

Listing 2.2: Overloading if statements

DSLs in LMS are organized via traits that are later mixed together via mixin composition [OZ05]. By convention, the traits are separated into "interface", "core implementation", "optimization" and "code generation" traits. Interface traits typically end on `-Ops`, core implementation traits on `-Exp` and optimization traits on `-Opt`. Code generation traits are prefixed by their target language, e.g. `ScalaGen-` for Scala code[1].

The trait `Base`, provided by LMS, defines the abstract type constructor [MPO08] `Rep[+T]`, and is used to distinguish between binding times. A type of `Rep[Int]` represents a staged value that will evaluate to an integer expression in the next stage.

Assuming that there are staged equivalents of all used constructs like numeric operations etc., LMS allows it to stage a program simply by annotating the correct types with the `Rep` type constructor. It is noteworthy that all operations on own types, that are staged, have to be defined by the programmer. In case of a large project, the overhead of writing the staged versions for all operations is therefore quite large.

The result of a program written in the DSL is a graph of intermediate representation (IR) nodes, representing the original program, that was expressed using the provided interface. LMS comes with predefined IR nodes for standard Scala operations together with the relevant Scala code generators and in some cases also for the CUDA platform as well as other languages. Assuming that the traits are organized according to the conventions, clients of a DSL have no access to the internal representation as IR nodes. This allows the DSL author to modify the internal representation, for example by replacing IR nodes with an optimized version. Additionally, the graph of IR nodes can be annotated by the programmer via the use of normal Scala methods, shown in listing 2.3:

```scala
def syms(e: Any): List[Sym[Any]]              // value dependence (must before)
def softSyms(e: Any): List[Sym[Any]]          // anti dependence (must not after)
def boundSyms(e: Any): List[Sym[Any]]         // nesting dependence (must not outside)
def symsFreq(e: Any): List[(Sym[Any], Double)] // frequency information (classify
                                              // sym as hot, normal or cold)
```

Listing 2.3: Graph Relations, from [Rom12, p. 71]

These methods create annotations in the IR graph, which serve as a guide for the built in code motion and allow the programmer to restrict the extent of allowed transformations, for example to fix a loop internal variable to the inside of the enclosing expression. By default, LMS assumes that expressions are pure in a functional sense (i.e. side-effect free). To mark operations that have side effects, LMS provides a method called `reflectEffect`, together with some alternatives [Rom12, p. 76]. Operations enclosed by `reflectEffect` are tracked and serialized automatically by the framework.

Figure 2.1 shows a minimal example for a DSL in LMS implementing simple integer addition via the + operator on values of type `Rep[Int]`. In addition to the basic DSL we introduce one optimization trait to optimize addition with 0.

[1] Our naming conventions differ slightly in so far, that we have a `-Syntax` suffix for interface traits and use the `-Ops` suffix to denote the core implementation trait. We find that these names make it more clear what the purpose of a trait is when compared to the standard conventions

```scala
trait Addition {
  def infix_+(lhs: Rep[Int],
    rhs: Rep[Int]) = addition(lhs,rhs)

  def addition(lhs: Rep[Int],
    rhs: Rep[Int]): Rep[Int]
}
```

(a) Interface

```scala
trait AdditionExp extends Addition {
  case class Addition(lhs: Rep[Int],
    rhs: Rep[Int]) extends Def[Int]

  override def addition(lhs: Rep[Int],
    rhs: Rep[Int]) = Addition(lhs,rhs)

}
```

(b) Core implementation

```scala
trait AdditionExpOpt extends AdditionExp {
   override def addition(lhs: Rep[Int],
     rhs: Rep[Int]) = (lhs,rhs) match {
       case (Const(0), y) => y
       case (x, Const(0)) => x
       case _ => super.addition(lhs,rhs)
     }
}
```

(c) Optimization

```scala
trait ScalaGenAddition {
  val IR: AdditionExp; import IR._

  override def emitNode(sym: Sym[Any],
    rhs: Def[Any]) = rhs match {
    case Addition(a,b) =>
      emitValDef(sym,
        quote(a) + " + " + quote(b))
    case _ => super.emitNode(sym, rhs)
  }
}
```

(d) Code generation

Figure 2.1: DSL for integer addition

2.4 Delite

Delite provides "compilation and runtime support for execution [...] to the DSL developer" [RSL+11] on top of LMS, together with predefined parallel execution templates, that will generate heavily optimized code, according to the template. Using LMS, the programmer has to take care of the compilation and execution of generated code himself. This is not the case in Delite, which provides a separate compilation step that generates so called *kernels*, units of generated code, while traversing the graph of IR nodes. During the traversal, Delite also keeps track of interdependencies between kernels and generates the "Delite Execution Graph" (DEG) which captures a list of in- and outputs together with the discovered interdependencies. Finally, the DEG together with static machine details, like number of available CPUs/GPUs, is used to create an execution plan for the target hardware. The plan itself consists of kernel calls with the necessary synchronizations.

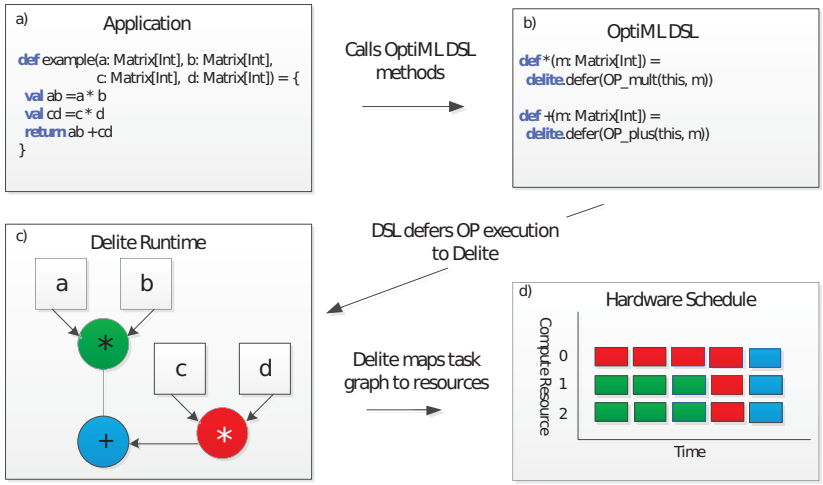


Figure 2.2: Execution of a Delite application, from [CSB+11, p. 5]

Figure 2.2 shows an example application execution using Delite. The two matrix multiplications are scheduled to be computed in parallel by the scheduling algorithm.

3 Design and Implementation of OptiReactive

This chapter presents the design and implementation of our DSL, called OptiReactive. Firstly, we present SimpleReactive, a simple base library for reactive programming in Scala. Afterwards we present OptiReactive, a DSL that fuses multistage and reactive programming.

3.1 Notations and Conventions

From this point on, we will make use of so called *dependency graphs* to illustrate the structure of reactive programs. A dependency graph is a directed graph, that typically has no cycles. *Nodes* in the graph represent reactives like `ReactiveVars` or `Signals`, *directed edges* denote the direction of the dependency, resulting in a "depends on" relation between the corresponding nodes. Different node types, representing e.g. `ReactiveVars` or `Signals`, are distinguished by shape and color of the node. `ReactiveVars` only occur as leaves, while `Signals` may occur as inner nodes or leaves. If there is a directed edge between two nodes, this means that the origin of the arrow depends on the value of the node the arrow points to. Figure 3.1 shows an example of a simple code snippet together with its dependency graph.

```
1 val variable = ReactiveVar(1)
2 val s1 = Signal(variable) { variable.get + 1 }
3 val s2 = Signal(variable) { variable.get * 2 }
```

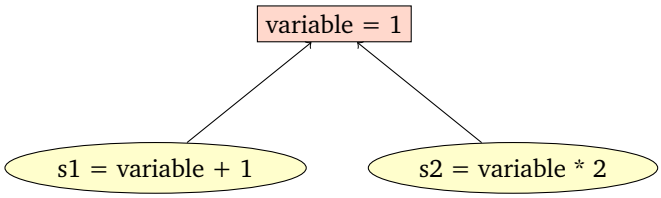


Figure 3.1: Example program with resulting dependency graph

Inside of nodes, we will use the form

$$identifier = expression$$

The *identifier* defines a new name that can be used inside the label of other nodes which depend on the one with the identifier inside. *expression* represents some kind of Scala expression, with the simplification that we omit the explicit `get` or `apply` call that is necessary to obtain the current value of a reactive. A slightly different form:

$$identifier = \{exp1, exp2, ..., expN\}$$

will be used to denote that *identifier* is set to *exp1*, *exp2* etc. in the given order.

3.2 SimpleReactive

As the back end of our DSL, we use a simplified library for reactive programming, named `SimpleReactive` which is modified from an initial base version, contributed by Dr. Guido Salvaneschi.

`SimpleReactive` provides two kinds of time-varying values: `Signals` and `ReactiveVars`. Both represent computations that may depend on other time-varying values. A `ReactiveVar` is different from a `Signal` in the sense, that it provides methods to directly change the value inside it. Due to their mutability, `ReactiveVars` may only occur as a root in dependency graphs. In the rest of this work we assume that the expressions inside of `Signals` or `ReactiveVars` are *pure* in the functional sense, i.e. their stored expression have no side-effects. We will use the more general term *reactives* to relate to both, `ReactiveVars` and `Signals`.

Figure 3.2 shows the class hierarchy used in `SimpleReactive`. From a high level point of view, we divide between two types, *dependency holders* and *dependents*. Internally, reactivity is implemented via the Observer pattern [GHJV94]. Nodes are subjects and observers, allowing them to communicate with their dependency holders on which they depend, as well as notifying their dependents if changes occur.

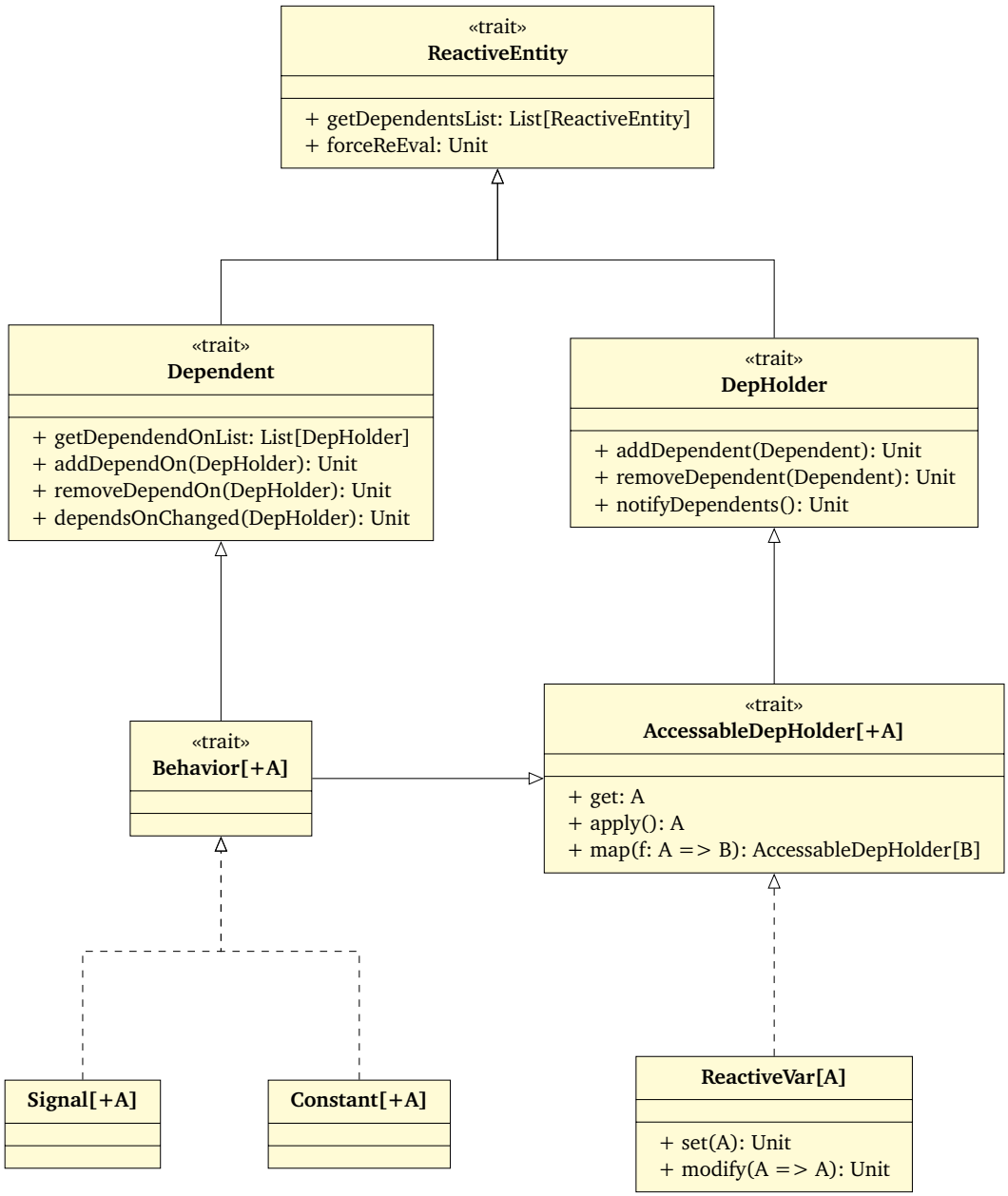


Figure 3.2: Design of SimpleReactive

Anything that may "hold" a value of interest for dependents is represented by the `DepHolder` trait. `DepHolders` provide methods for dependents to register themselves, as well as a method that is called to notify dependents about events.

Entities, that somehow depend on a value of a `DepHolder` are instances of the `Dependent` trait. `Dependents` internally track the `DepHolders` they depend on and provide methods to modify this list. Lastly, they expose a method that is triggered if a change occurs.

The `AccessableDepHolder` trait extends `DepHolders` with a type parameter, methods to access the stored expression's result and the `map` function, that applies a function to the `AccessableDepHolder` and creates a new version.

Up until now, dependents and dependency holders were separate concepts. As `ReactiveVars` are only allowed to occur as a root in dependency graphs, they only implement the `AccessableDepHolder` trait, adding mutator methods. `Signals` however, can actually be dependents and dependency holders. Trait `Behavior` captures this fact by extending both, `AccessableDepHolder` and `Dependent`. Concrete implementations of `Behavior` are `Signal` and `Constant`, representing a time-varying pure expression and a constant expression that will never change, respectively.

3.3 OptiReactive

OptiReactive, named in the same scheme as other Delite DSLs (see section 6.2.3), is our DSL built on top of Lightweight Modular Staging (LMS) [RO10] and Delite [Rom12, RSL+11]. The main part of OptiReactive does not depend on Delite, the only optimization that requires it, is the parallel propagation described in section 4.3.3. First we give an overview of the design. The next section concentrates on the implementation and in the last we walk through a complete example of a program written in OptiReactive.

3.3.1 Design

The type hierarchy used in OptiReactive is the same as in SimpleReactive (Figure 3.2), with the difference, that OptiReactive uses only abstract classes and traits. Types are only required to guide the Scala compiler regarding allowed subtype relations, e.g. treating a `Signal` as a `DepHolder`, when using the DSL. This decouples OptiReactive from the concrete base library we will generate code for. Therefore, changing from our base library SimpleReactive to another implementation of reactive programming is easy under the condition that the new implementation is also based on abstractions like `ReactiveVars` and `Signals`.

Figure 3.3 shows the trait hierarchy of OptiReactive's components. Nodes filled with green are nodes that optimize some part of the DSL, or perform syntactic transformations.

The `Reactivity` trait defines the interface part of our DSL, by integrating all of our syntax traits into one. `Reactivity` therefore represents the complete visible interface available to clients of the DSL. In actual programs, the `Reactivity` trait is commonly mixed in with at least one other trait that lifts standard Scala constructs into the DSL. We therefore define another trait, `ReactiveDSL`:

```
trait ReactiveDSL extends Reactivity with ScalaOpsPkg with LiftScala
```

`ScalaOpsPkg` provides operations for standard Scala constructs, `LiftScala` enables implicit lifting of unstaged values, i.e. from `Int` to `Rep[Int]`.

Trait `ReactivityExp` combines the various core implementation traits, that override the abstract methods of the corresponding interface traits. The trait `ReactivityExpOpt` extends the core implementation with the constant folding optimization and includes the trait that replaces all standard `map` calls with a fused version. `ScalaGenReactivity` integrates the code generation traits, required for the conversion of IR nodes to actual code. `ScalaGenReactivityOpt` extends on the previous trait, including an alternative version of code generation that applies the constant folding optimization.

3.3.2 Implementation of reactive abstractions

Listing 3.1 shows the traits implementing `Signals` in OptiReactive. For simplicity, we omit the concrete implementation of the `map` method.

The trait `SignalSyntax` defines a method `apply` on the `Signal` object, that can be called as

```
Signal.apply(dhs)(f)
```

which is equivalent to the syntax used in SimpleReactive. Internally, the call is forwarded to an abstract method, `new_behavior` (line 6), which produces a value of type `Behavior` as the result. In trait `SignalOps`, line 9, this method is overridden with a concrete implementation, creating a `SignalCreation` IR node. In `ScalaGenSignals` (line 24) we

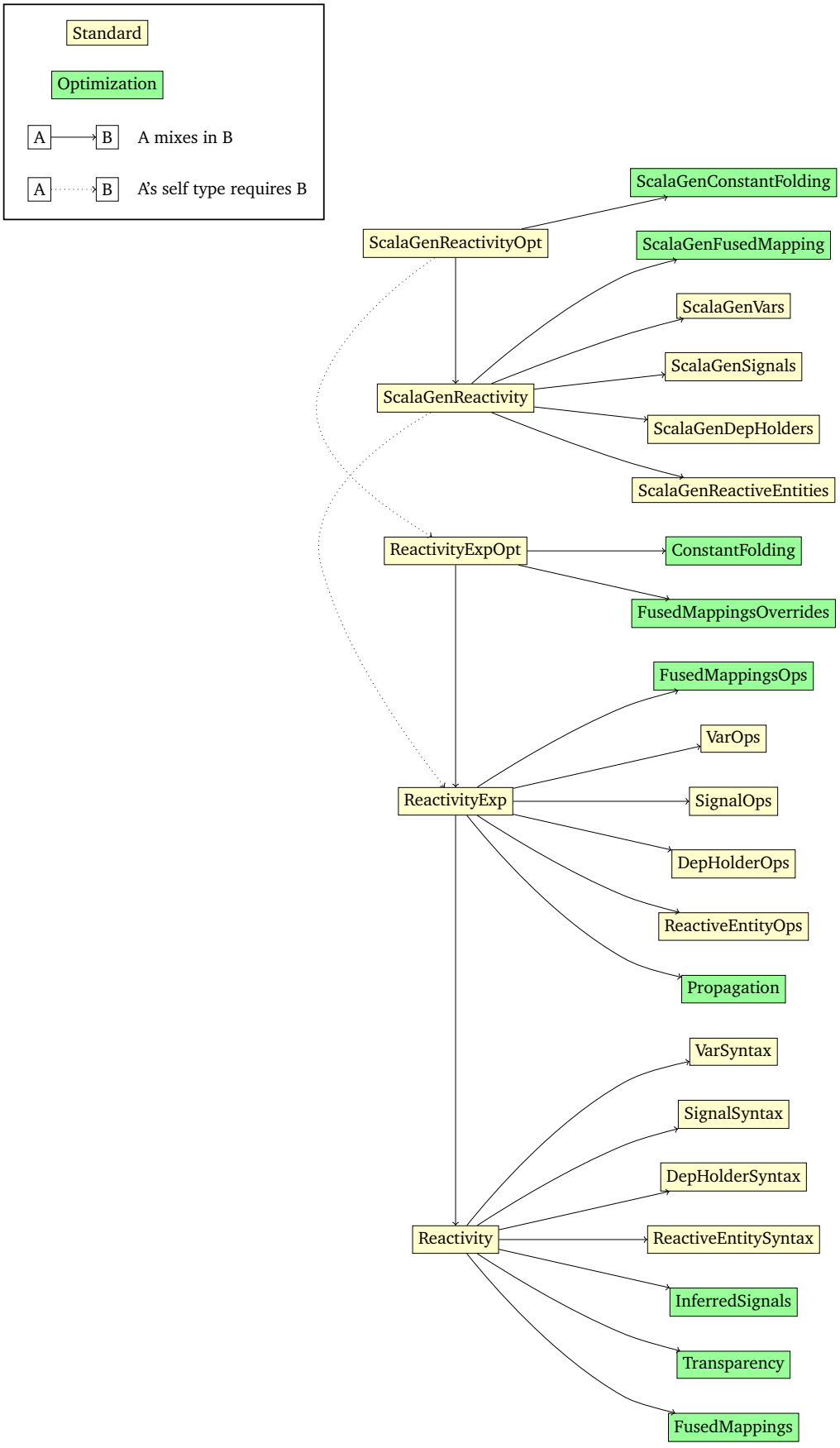


Figure 3.3: OptiReactive's trait hierarchy

```
trait SignalSyntax extends Base {
  object Signal {
    def apply[A:Manifest](dhs: Rep[DepHolder]*)(f: => Rep[A]) = new_behavior(dhs, f)
  }

  def new_behavior[A:Manifest](dhs: Seq[Rep[DepHolder]], f: => Rep[A]): Rep[Behavior[A]]
}

trait SignalOps extends EffectExp with FunctionsExp {
  this: SignalSyntax =>

  case class SignalCreation[A:Manifest](dhs: Seq[Exp[DepHolder]], body: Block[A])
      extends Def[Behavior[A]]

  override def new_behavior[A:Manifest](dhs: Seq[Exp[DepHolder]],
    f: => Exp[A]): Exp[Behavior[A]] = SignalCreation(dhs, reifyEffects(f))

  override def boundSyms(e: Any): List[Sym[Any]] = e match {
    case SignalCreation(dhs,body) => effectSyms(body)
    case _ => super.boundSyms(e)
  }
}

trait ScalaGenSignals extends ScalaGenReactiveBase with ScalaGenFunctions {
  val IR: SignalOps
  import IR._

  override def emitNode(sym: Sym[Any], node: Def[Any]): Unit =  node match {
    case SignalCreation(dhs,f)  => emitValDef(sym,
      simpleReactivePkg + "Signal(" + dhs.map(quote).mkString(", ") + ") { ")
        emitBlock(f)
        stream.println(quote(getBlockResult(f)) + "\n")
      stream.println("}")
    case _ => super.emitNode(sym,node)
  }
}
```

Listing 3.1: Implementation of Signals in OptiReactive

generate the necessary code to create an actual `Signal`. In lines 18 and 19, we specify that any effectful statements inside the `body` are bound. Together with the `reifyEffects` method in line 16, this makes sure that the generated code for the body of a `Signal` is not hoisted out by code motion.

The generated code is where OptiReactive uses the underlying base library. In this case the library is SimpleReactive, but by changing the code generation for the IR nodes, OptiReactive can support other implementations as well.

3.3.3 A complete example

```
1  package dsl.reactive
2
3  trait ExampleProgram extends ReactiveDSL {
4    def main() = {
5      var variable = 1
6      val signal1 = Signal(variable) { variable.get + 1 }
7      val signal2 = ISignal { variable.get + 1 }
8
9      println(signal1.get) // => 2
10     println(signal2.get) // => 2
11
12     variable = 5
13
14     println(signal1.get) // => 6
15     println(signal2.get) // => 6
16   }
17 }
```

Listing 3.2: OptiReactive example code

In this section, we will walk through an example program, written in OptiReactive. Listing 3.2 shows the program as written by the developer, while listing 3.3 shows the program after replacing the DSL syntax with the underlying method calls. Finally, listing 3.4 shows the program after the method calls are replaced with the resulting IR nodes. In the following, we go through these transformation step by step.

Considering only the body of the `main` method in listing 3.2, the code is almost identical to SimpleReactive, making it easy to port code from one language to another. In lines 5 and 12 we assume that the `Transparency` trait is mixed in, which enables the overloading of variable assignment as will be further described in section 4.2.1. Line 7 demonstrates the implicit dependency inference, further discussed in section 5.1. For simplicity, we will omit the type signatures, therefore hiding the fact that the actual types are wrapped inside a `Rep` type from the DSL clients point of view.

In the first step, the example code from listing 3.2 leads to a row of method calls to the internal helper methods of the DSL. Listing 3.3 shows the same program, using the explicit method calls. Instead of creating a Scala `var` in line 5, we create a regular `val` with the reactive `ReactiveVar` as the right hand side. This works due to Scala-Virtualized, which enables us to intercept the var creation. The same goes for the variable assignment in line 12, which is replaced by our DSL method `dep_holder_set`. In the next step, the method calls are evaluated and replaced by IR nodes, represented by case classes [OAC+04]. In the interface traits, these methods are kept abstract. The actual implementation is defined in implementation traits, which are not visible to clients. In general, these methods return an IR node, representing a computation that will yield the expected result in a later stage.

Listing 3.4 shows the example again with the corresponding IR nodes that result from the method calls in listing 3.3. Note that the dependencies of `signal2` in line 7 were inferred between these steps, assuming the corresponding trait (described in section 5.1) was mixed in.

In lines 6 and 7, we use a LMS provided method `reifyEffects` to mark the expression stored as the second argument in `SignalCreation` nodes. By doing this, we disallow the built-in code motion to hoist out the expression during code generation. In line 12 we use `reflectEffect` because we are mutating a `ReactiveVar` and have to make sure that this effect is serialized properly.

Finally, LMS will use the supplied code generators to generate code from the IR nodes. This is done via the method `emitNode`, defined in a base code generator trait, by pattern matching on IR nodes and emitting corresponding code.

```scala
package dsl.reactive

trait ExampleProgram extends ReactiveDSL {
  def main() = {
    val variable = new_reactive_var(1)
    val signal1 = new_behavior(variable)(variable.get + 1)
    val signal2 = new_inferred_behavior(variable.get + 1)

    println(dep_holder_acess(signal1)) // => 2
    println(dep_holder_access(signal2)) // => 2

    dep_holder_set(variable,5)

    println(dep_holder_access(signal1)) // => 6
    println(dep_holder_access(signal2)) // => 6
  }
}
```

Listing 3.3: OptiReactive example with explicit method calls

```scala
package dsl.reactive

trait ExampleProgram extends ReactiveDSL {
  def main() = {
    val variable = VarCreation(1)
    val signal1 = SignalCreation(variable, reifyEffects { variable.get + 1 })
    val signal2 = SignalCreation(variable, reifyEffects { variable.get + 1 })

    println(AccessDepHolder(signal1)) // => 2
    println(AccessDepHolder(signal2)) // => 2

    reflectEffect(SetDepHolder(variable,5))

    println(AccessDepHolder(signal1)) // => 6
    println(AccessDepHolder(signal2)) // => 6
  }
}
```

Listing 3.4: OptiReactive example with IR nodes

```scala
override def emitNode(sym: Sym[Any], node: Def[Any]): Unit =  node match {
  case VarCreation(v) => emitValDef(sym, "Var(" + quote(v) + ")")
  case _ => super.emitNode(sym,node)
}
```

Listing 3.5: Code generator for VarCreation

In addition to the code generator for `Signals` in listing 3.1, listing 3.5 shows a simpler code generator for `VarCreation` IR nodes. Every code generator uses a catch all pattern to delegate up the hierarchy in the case that it cannot handle the given IR node. In case that no code generator can handle the IR node, there will be an exception at runtime.

4 Optimizations

OptiReactive enables several program transformations. In this chapter we describe the transformations based on LMS and Delite, implemented in this thesis. First we describe syntactic transformations that makes reactive programming easier to use. Then we present optimizations that enable performance improvements. All but one of the following transformations are applicable to OptiReactive using only LMS. The exception is “Parallel propagation”, which makes use of the Delite built-in execution patterns.

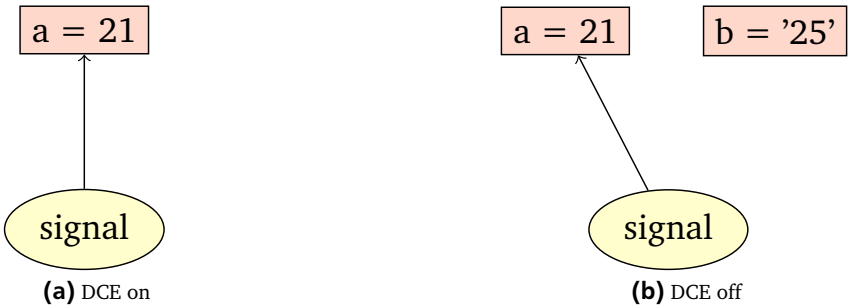
4.1 Built-in optimizations of LMS and Delite

By using the LMS and Delite frameworks for our DSL, we get some of the classic compiler optimizations that work on the DSL level. By default, *common subexpression* & *dead code elimination*, *constant folding*, as well as *code motion* [ALSU06] is supported. The latter is of restricted use in our DSL as we have to guarantee that expressions inside of reactives are *not* hoisted out.

Dead code elimination (DCE) can be quite beneficial in reactive applications, because the indirection introduced by the DSL implementation of reactive abstractions make it hard for the compiler to apply DCE at the code level. By using OptiReactive, we gain full support of DCE at the DSL level: reactive values like `Signals` or `ReactiveVars`, that are used in the DSL, but are unreachable after evaluating the first stage of the program, are guaranteed to be removed in the resulting code. Listing 4.1 shows an example Scala program. The resulting graph structures with and without dead code elimination are shown in figure 4.1.

```
1  var condition: Boolean = true
2  val a = ReactiveVar(21)
3  val b = ReactiveVar("25")
4
5  val signal = ISignal {
6    if (condition) {
7      "Value: " + a.get.toString
8    } else {
9      "Value: " + b.get.toString
10   }
11 }
```

Listing 4.1: Dead code elimination example



(a) DCE on **(b)** DCE off

Figure 4.1: Dependency graph with DCE turned on/off

The advantage of removing reactives is that it is not necessary to update them during propagation through the dependency graph, which can considerably speed up the execution of the program. The elimination in listing 4.1 can work in this case, because the condition is evaluated in a prior stage and is therefore statically known, so that the analysis sees only one of the two branches. If the condition is a staged boolean, `signal` depends on both `a` and `b`, inhibiting DCE.

While this is only a small example, in a more sophisticated application there often occur situations where multiple of these reactive values or whole paths can be eliminated.

Other built-in optimizations like constant folding apply to expressions inside `Signals`, but do not work at DSL level by default (we will develop a working version in section 5.2). Common subexpression is of limited use in our implementation, because updating the dependency graph is a side-effecting operation, which is marked as impure for the framework and can therefore not profit from this particular optimization in most cases.

4.2 Syntactic transformations

This section presents syntactic transformations, that can be applied by OptiReactive. We will look at an approach to transparent reactivity in Scala and show OptiReactive's solution for dependency inference.

4.2.1 Transparent reactivity

Scala-Virtualized enables the overloading of selected constructs in Scala. This makes it possible to override control structures as well as object creation and enables the addition of methods to existing types. When using a reactive library, there is syntactic overhead to overload or convert existing mechanisms so that they interact with the reactive constructs. In our DSL, `ReactiveVar` is such an example. The code snippet in figure 4.2 (a) shows a simple example program using only functionalities provided by SimpleReactive.

```
1 class Point(x: Int, y: Int) {
2   val xPos = ReactiveVar(x)
3   val yPos = ReactiveVar(y)
4 }
5
6 def swap(point: Point) {
7   point.xPos.set(point.yPos.get)
8   point.yPos.set(point.xPos.get)
9 }
```

(a) Using explicit DSL syntax

```
1 class Point(x: Int, y: Int) {
2   var xPos = x
3   var yPos = y
4 }
5
6 def swap(point: Point) {
7   point.xPos = point.yPos.get
8   point.yPos = point.xPos.get
9 }
```

(b) Using Scala-Virtualized

Figure 4.2: Example using explicit DSL syntax and Scala-Virtualized

Using the override mechanisms of Scala-Virtualized, we can write code that looks more like plain Scala, while still having the same semantics as the previous program. Figure 4.2 (b) shows the example again using overloaded var creation and assignment.

The programmer does not have to know anything about the specific methods that are used to guarantee that the reactive framework detects the changes and dependent reactives are updated accordingly.

Unfortunately we cannot overload val assignment with `Signal` expressions in the same manner. It would be possible to use explicit type annotations on the left to trigger an implicit conversion ([OAC+04]). The right side of the assignment, however is still reduced to an evaluated form, while we actually want the unevaluated expression. Assuming that this missing feature is added to Scala-Virtualized, it becomes possible to write almost regular Scala code, that will have reactive semantics, when used with the enabled overloadings. The idea of transparent reactivity is also mentioned in [BCK07].

4.2.2 Dependency Analysis

Both SimpleReactive and OptiReactive use `ReactiveVars` and `Signals` as the main reactive abstractions. `Signals` are constructed via *Signal Expressions*, as in Scala.React [MRO10]. When creating `Signals`, the dependencies have to be either explicitly specified or inferred. The `Signal.apply` method in Scala.React, responsible for discovering the dependencies, makes use of a thread local stack, to track the concrete dependents. A simple alternative is to explicitly pass the dependencies into the `Signal.apply` method, when creating the Signal. The explicit approach has the disadvantage, that it transfers the responsibility to the programmer and in case of incorrect dependencies, the resulting code may not behave as expected. Furthermore there may be computations at staging time that transform the code of the expression in ways that are not easy to analyze in advance. Conservatively passing in more dependencies than needed will additionally inhibit the dead code elimination and may therefore lead to a bigger dependency graph, which the underlying framework has to maintain, typically manifesting in reduced runtime performance.

Our base library, SimpleReactive, uses the explicit approach when creating `Signals` via the `Signal.apply` method, similar to Scala.React, but with an additional argument for dependencies. OptiReactive on the other side, extends the syntax with another form of Signal expressions, defined as `ISignal.apply`, that is able to infer the required dependencies from the given expression and is syntactically equivalent to the one in Scala.React. The inferred dependencies are subsequently passed into the Signal expression form that takes explicit dependencies as an argument.

As an example of how this works, consider the example Scala program in listing 4.2 with the resulting dependency graph, shown in figure 4.3.

```
1 val v1 = ReactiveVar(5)
2 val v2 = ReactiveVar(10)
3 val v3 = ReactiveVar(15)
4
5 val s1 = ISignal { v1.get + v2.get }
6 val s2 = ISignal { v2.get + v3.get }
7
8 val s3 = ISignal { s1.get * s2.get }
```

Listing 4.2: Dependency analysis: example code

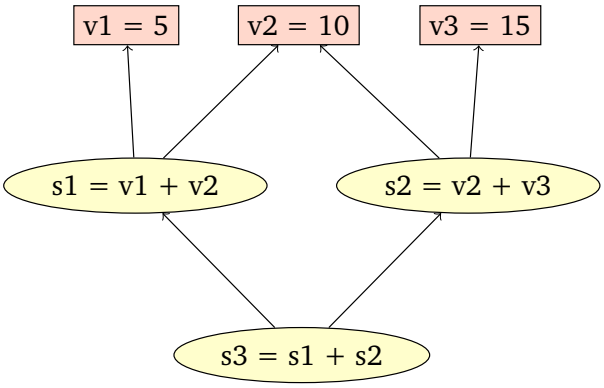


Figure 4.3: Result of dependency analysis

The `apply` or `get` method, when invoked on a reactive value, will result in an `AccessDepHolder` IR node. Detecting the dependencies on reactive values can be done by analyzing the unevaluated expression, passed in to the `Signal.apply` factory method after converting it to a `Block`. `Block` is provided by LMS and encapsulates a lambda that takes no arguments. Additionally, LMS provides a method to find all symbols that a given object references. Applying this method to the `Block` yields a list of `Syms` which we can search for IR nodes representing the access to reactive values, i.e. `AccessDepHolder`. The LMS trait `Expressions` provides a method to find definitions given a `Sym`:

```
findDefinition: Sym[T] => Def[T]
```

Applying this method to the list of detected `Syms` of the `Block` yields a list of dependency holders. This final list can be passed into SimpleReactive's Signal expression form, to create a `Signal`. Due to the fact that the analysis is done at staging time, it has no impact on the runtime performance.

In case of dynamic dependencies, the analysis infers conservatively. For example when using if-expressions inside a `Signal` expression, there are two different cases. In figure 4.4 (a) the condition of the if-expression is an unstaged boolean, therefore it will be evaluated in the intermediate stage and our dependency analysis only sees the branch that was taken. If the condition is a staged value, as in figure 4.4 (b), our dependency analysis will be conservative and infer both `v1` and `v2`.

```scala
val cond: Boolean = true

val v1 = ReactiveVar(1)
val v2 = ReactiveVar(2)

ISignal { if (cond) v1.get else v2.get }
```

(a) Unstaged condition

```scala
val cond: Rep[Boolean] = true

val v1 = ReactiveVar(1)
val v2 = ReactiveVar(2)

ISignal { if (cond) v1.get else v2.get }
```

(b) Staged condition

Figure 4.4: Two alternatives of if-expressions

4.3 Performance optimizations

This section presents optimizations applied by OptiReactive, that improve the performance of the program. The presented optimizations are constant folding, map fusion and parallel propagation.

4.3.1 Constant folding

Reactive libraries may require the use of a reactive value, while the client wants to provide a constant. The common solution is to introduce a special reactive that does not change, therefore lifting constant values. Wrapping constant values introduces overhead in terms of memory and in terms of propagation performance as well, depending on the concrete implementation. This is due to the fact that reactives that depend on constant values are included in the dependency graph, which the library has to maintain over the course of a program execution.

In OptiReactive, we can make use of the DSL IR nodes to resolve these issues. A `Signal` for which there were no dependencies found during dependency analysis (section 4.2.2), or if the explicitly supplied dependencies list is empty, OptiReactive will replace the `SignalCreation` IR node with a `ConstantCreation` node, invisible for the client of the DSL. To summarize, we can replace a `Signal` with a `Constant`, if one of the following is true:

(1) the list of explicit/inferred dependencies is empty

(2) all of the specified dependencies are constant themselves

Figure 4.5 shows a example program together with the resulting dependency graph. In the program we can replace *all* `Signals` with constants, because `p1` and `p2` are constants according to (1), `dist` and `sqDist` according to (2).

```scala
def euclidDistance(p1: Point,
  p2: Point): Double = ???

val p1 = ISignal { Point( 5, 5) }
val p2 = ISignal { Point(10, 5) }

val dist = ISignal {
  euclidDistance(p1.get,p2.get)
}

val sqDist = ISignal {
  dist.get * dist.get
}

println(sqDist.get)
```

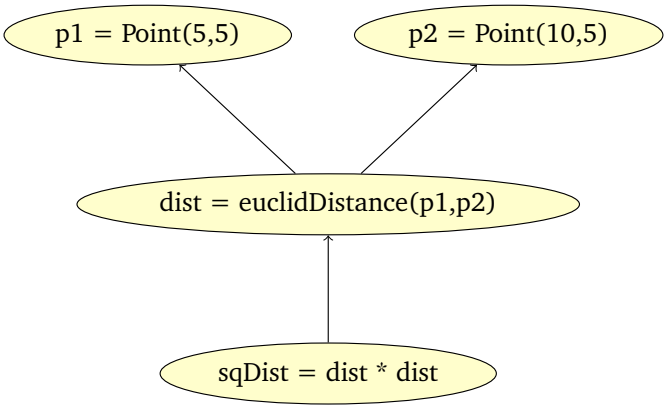


Figure 4.5: Constant conversion example program with dependency graph

What OptiReactive does, is that it replaces the `SignalCreation` IR nodes with `ConstantCreation` nodes, if the conversion described above was applicable. When generating code for constants, we can exploit this knowledge: instead of an explicit access of the constant via `get` or `apply`, we can "unwrap" the expression inside the `ConstantCreation` IR node and insert it instead of the explicit access call. Of course, this requires, as stated before, that expressions inside of reactives are pure.

Listing 4.3 shows a semantically equivalent snippet of code that will result from the transformations of the program shown in figure 4.5. If the Delite version of OptiReactive is used, the code will be spread out into kernels, but remain semantically equivalent. The code shows no trace of reactives at all, because all of them are converted into constants and the access is unwrapped during code generation.

```
1 val p1 = Point( 5, 5)
2 val p2 = Point(10, 5)
3
4 val dist = math.sqrt(math.pow(p1.x - p2.x,2) + math.pow(p1.y - p2.y,2))
5 val sqDist = dist * dist
6
7 println(sqDist)
```

Listing 4.3: Constant conversion result

4.3.2 Map fusion

A common operation on `Signals` is to use the higher-order `map` function. Mapping over a `Signal` generates a new `Signal`, which applies the given function to its expression at each evaluation. In general, every use of the `map` function leads to a new `Signal` being instantiated and registered in the reactive framework. In some cases, the intermediate `Signals` are not of interest at all and slow down the execution of the final program.

OptiReactive introduces a modified version of `map`, called `fuseMap`, that does not generate intermediate structures. By using a special trait, the standard `map` method can be overridden to use the fused version. The fusing works by replacing the IR node representation used in OptiReactive. Without the optimization, calling the `map` function on a `Signal` creates an `MappedBehavior` IR node, storing the function we are mapping, as well as the behavior we are mapping over. Listing 4.4 shows an example where `map`-calls on a signal are chained.

```
1     signal.map(i => i + 1).map(i => i + 2).map(i => i + 3)
```

Listing 4.4: Sequence of map calls

```
1 MappedBehavior(i => i + 1, signal)}
2
3 MappedBehavior(i => i + 2,
4   MappedBehavior(i => i + 1, signal))}
5
6 MappedBehavior(i => i + 3,
7   MappedBehavior(i => i + 2,
8     MappedBehavior(i => i + 1, signal)))}
```

(a)

```
1 MappedBehavior((i => i + 1, signal)
2
3 MappedBehavior((i => i + 2) compose
4                (i => i + 1, signal)
5
6 MappedBehavior((i => i + 3) compose
7                (i => i + 2) compose
8                  (i => i + 1, signal)
```

(b)

Figure 4.6: IR nodes with unfused (a) and fused (b) mapping

The resulting IR nodes of each intermediate `map`-call are shown in figure 4.6 (a). The unfused version results in nested `MappedBehavior` nodes. In this form, code has to be generated for each IR node, because the outermost depends transitively on all inner ones.

OptiReactive's map fusion avoids the creation of intermediate nodes by accumulating the function that will be mapped via function composition. Instead of wrapping every `map` call with `MappedBehaviors` we accumulate the function that will be applied at the end as the first argument to the IR node. The repeated `map` calls from listing 4.4 therefore result in IR nodes of the form as shown in figure 4.6 (b), where `compose` is the standard Scala function composition in staged form. The actual code generation for the resulting IR node is unchanged from the original, with the addition of a simple generator for the staged function composition. The result is that no intermediate reactives are created. Instead the function is accumulated via composition and finally used as the argument to the standard `map` function of the underlying libray, in our case SimpleReactive.

4.3.3 Parallel propagation

OptiReactive can optionally generate code to propagate changes through the dependency graph at the DSL level, in addition, or as a replacement for the propagation that may be done by the base library. The advantage is that the built-in Delite execution patterns for parallel operations can be used, offering high level abstractions and loop fusion. In this section we investigate two approaches on how to make use of this infrastructure. We concentrate on propagation using the DSL abstraction and assuming that the base library does no propagation by itself.

When a reactive value changes, for example via the `ReactiveVar.set` method, all dependents of this reactive have to be notified. Performing this in parallel is straightforward: simply notify each dependent in parallel. Listing 4.5 shows a concrete implementation for the built in Delite execution pattern `DeliteOpForeach`.

```
1 case class NotifyDependents(dh: Exp[ReactiveEntity])
2     extends DeliteOpForeach[ReactiveEntity] {
3   def func: Exp[ReactiveEntity] => Exp[Unit] = _.reEvaluate()
4   val in: Exp[DeliteCollection[ReactiveEntity]] = dh.getDependents
5   val size: Exp[Int] = dh.getDependents.size
6   def sync: Exp[Int] => Exp[List[Any]] = _ => List[Any]()
7 }
```

Listing 4.5: DeliteOpForeach for parallel execution

In line 1 we declare a new IR node called `NotifyDependents`, which extends from `DeliteOpForeach`, parameterized over the collection element type, `ReactiveEntity`. The four necessary definitions (line 2-6) are:

`func` the function that is applied to each element

`in` the collection to iterate over which is acquired by requesting all dependents of the reactive

`size` the number of elements

`sync` a list of `Sym`s that have to be synchronized

With this IR node defined, we can extend the definition of IR nodes that should invoke a propagation, e.g. for `ReactiveVar.set`. `NotifyDependents` itself propagates one level, the remaining task is to apply it so that the whole graph is updated.

A simple, recursive algorithm would involve the following steps:

- external detection that a propagation is necessary
- reevaluate the expression of the node
- retrieve a list of all dependents affected by the change
- recursive application to all dependents

This is straightforward to formulate in Delite. Unfortunately, the runtime does not (yet) support the execution of recursive methods[1]. This is due to the fact, that Delite supports heterogeneous hardware (GPUs,CPUs...), which makes it complicated to support recursion, as it might not be supported on all types of target hardware. An implementation of the algorithm in an iterative form resulted in a version, that only supports propagation up to a finite depth, which has to be specified in advance. Problems with the Delite implementation made it not possible to generalize the algorithm to update the complete dependency graph.

We expect that this is a minor issue, and may already be fixed in some of the developer versions, which we were unable to test due to some incompatibilities regarding the framework. In the evaluation, we set the number of levels to propagate manually to the value that would be used by the generalized version.

[1] According to a conversation with Arvind Sujeeth on the Delite mailing list (delite-devel@googlegroups.com) on July, 2nd.

5 Evaluation

This chapter evaluates the functionalities presented in chapter 4. First, we discuss dependency analysis regarding usability and correctness. The rest of the chapter investigates performance improvements of constant folding, map fusion and the Delite version of OptiReactive. The results show, that the presented optimizations do improve the performance of the generated programs and that there is no significant performance overhead when using OptiReactive, compared to the base library, even if no optimizations are applicable.

The benchmarks were run on a Thinkpad X220 with an Intel dual core i7 2620M 2.7 GHz processor and 8GB of RAM. The code was compiled and executed with Scala 2.10.0-M1-virtualized, the OpenJDK Runtime Environment (IcedTea 2.3.10) and the OpenJDK 64-Bit Server VM (build 23.7-b01). The project branched from the official Delite 'optiml-beta' branch at commit 14f30dd and uses LMS in version 0.3.

Each of the following experiments was run at least 15 times. The first five runs were discarded and the mean of the remaining runs was taken as the final result.

5.1 Dependency Analysis

In most cases, explicit dependencies can be provided manually by the programmer. Besides the fact that automatic inference is less error prone, there are use cases where it is not possible to determine the actual dependencies in advance. Listing 5.1 shows an example where the programmer cannot explicitly provide the minimal set of dependencies manually, due to a computation at the intermediate stage. The explicit type annotations are only given to improve readability and would normally be omitted.

```
1  val reactives: Seq[Rep[AccessableDepHolder[Int]]] = collection.Seq( /*...*/ )
2
3  val sum: Rep[AccessableDepHolder[Int]] = ISignal {
4    def filtered: Seq[Rep[AccessableDepHolder[Int]]] =
5      reactives.filter(_ => util.Random.nextBoolean)
6
7    filtered.foldLeft(unit(0))((acc,r) => acc + r.get)
8  }
9
10 println(sum.get)
```

Listing 5.1: Advantage of dependency analysis

In line 1, we define a sequence of reactives. The signal `sum` (line 3) depends on a subset of the defined `reactives` sequence. If explicit dependencies were used, the programmer would have to state all reactives as dependencies, because he cannot know which or if any of the reactives will be filtered out. Dependency Analysis, however can infer the correct dependencies, because the filtering is done at staging time.

5.2 Constant folding

The impact on the performance of our constant folding optimization depends on the target program. The implicit removal releases clients of the reactive library to specify constant signals explicitly, if they use a reactive library that offers an API that requires reactives as e.g. the argument of a method. For example, consider a geometric reactive library, that offers a method to calculate the distance between two time-varying points. As a client, we only want to measure the distance between a fixed point (e.g. the screen center) and one time-varying point (e.g. the mouse cursor position). One possible method to address this is to introduce a special reactive for constant values. In OptiReactive, we can handle this situation without requiring the use of a special type of reactive by automatically detecting if a reactive is constant or not.

Listing 5.2 shows an example, where the method `manhattanDistance`, that may be provided by a library, only accepts reactive values. The signal `staticCenter` is detected as a constant. Therefore, during code generation, instead of explicitly querying the content via the `get` method, OptiReactive will insert the value, `Point(0,0)`. This reduces the cost

```
def manhattanDistance(
  point1: AccessableDepHolder[Point],
  point2: AccessableDepHolder[Point]
): AccessableDepHolder[Int] = ISignal {
  math.abs(point1.get.x - point2.get.x) + math.abs(point1.get.y - point2.get.y)
}

val staticCenter = ISignal { Point(0,0) }
val dynamic = ReactiveVar(Point(0,0))
val mouse = ReactiveVar(Point(10,0))

val semiStaticDistance = manhattanDistance(staticCenter,mouse)
val dynamicDistance = manhattanDistance(dynamic,mouse)
```

Listing 5.2: Calculating the distance between points

of calculating the time varying signal `semiStaticDistance`, because only `mouse` has to be queried. Calculating dynamicDistance requires the program to query both, `mouse` and `dynamic` for its value. The main benefits of this optimizations are:

- reactives are replaced by normal code
- the resulting code can be optimized further

The first point means, that the reactive framework has less reactives to maintain over the course of the program execution. The second point may lead to additional optimizations being applicable, which can speed up the execution further. This is due to the fact, that the code is scattered inside reactives, hiding the internal computation structure and providing only access to the result. Constant folding merges the computations back into one place, allowing individual optimizations to inspect the computation as a whole. As an example where this optimization can dramatically improve the performance, consider the program shown in listing 5.3.

```
val v1 = ISignal { 43 }
val v2 = ISignal { 43 }
val v3 = ISignal { 43 }

val s1 = ISignal { fib(v1.get) + fib(v2.get) + fib(v3.get) }
val s2 = ISignal { fib(v1.get) + fib(v2.get) + fib(v3.get) }
val s3 = ISignal { fib(v1.get) + fib(v2.get) + fib(v3.get) }

val result = ISignal { s1.get + s2.get + s3.get }
result.get
```

Listing 5.3: Evaluation example for constant folding

Without the optimization, the resulting program will have a dependency graph as shown in figure 5.1. As a result, the computation of `fib(43)` is done nine times, where actually one computation would suffice. With constant folding activated, the `Signals` are replaced with `Constants`, which in turn are eliminated during code generation. This allows LMS's built-in common subexpression elimination to detect, that the same pure computation occurs repeatedly and can be reduced to only *one* occurrence. The generated code, with activated constant folding is shown in listing 5.4

Instead of calculating `fib(43)` repeatedly, the result is calculated only once and summed up the required number of times. In fact, the above code is a more verbose version of "`9 * fib(43)`".

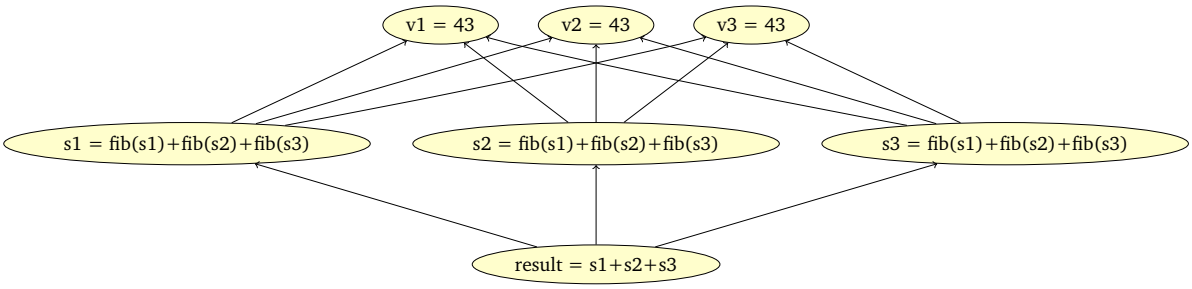


Figure 5.1: Dependency graph for constant folding example

```
1  val x3 = 43
2  val x4 = fib(x3)
3  val x5 = x4 + x4
4  val x6 = x5 + x4
5  val x8 = x6
6  val x9 = x8 + x8
7  val x10 = x9 + x8
8  val x12 = x10
```

Listing 5.4: Generated code with constant folding

5.3 Map fusion

To evaluate the performance improvements that result from the fusion of consequent `map` calls we consider an example program as shown in figure 5.2.

```
1  val v = ReactiveVar(43)
2  val f: Rep[Int] => Rep[Int] = x => fib(v.get)+x
3
4  val s1 = ISignal(fib(v.get))
5  val s2 = s1.map(f)
6  /* repeated calls of map(f) */
7  val sN = s2.map(f)
8
9  println(sN.get)
10
11 v.set(42)
12 println(sN.get)
```

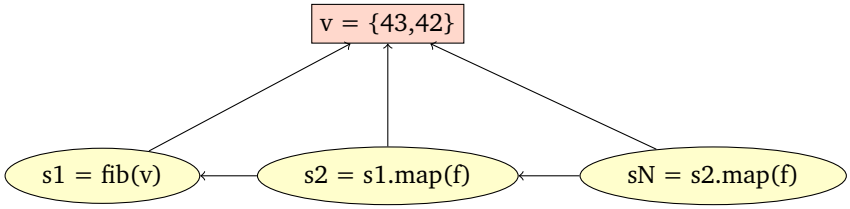


Figure 5.2: Example program for map calls with dependency graph

The general structure of the resulting dependency graph has a root in form of a `ReactiveVar`, initialized with `43` and one or more `Signals`. The signals have a dependency on the root and calculate the Fibonacci number at the specified position. After the first calculation is done, we change the root node's value to `42`, which triggers an update of all dependents. In the following, we measure the total run time of this program with a different number of chained `map` calls.

In figure 5.3 we show the performance of the previous example in figure 5.2, executed with an increasing number of maps. We see that the fused version is much faster than the unfused version. The reason is that without map fusion, each call to `map` results in an intermediate `Signal` being created and registered for updates. Therefore, when the value at the top of the dependency graph is modified, all intermediate `Signals` will be updated as well, although only the final result of the `map` calls is actually referenced. With our fusion optimization from section 4.3.2, we avoid the creation of these intermediate `Signals` by accumulating the function to map via regular function composition. The result is that none of the intermediate reactives are created, thereby reducing the overall memory consumption of the program and a change to the root node's value results only in one update of the final `Signal`'s value.

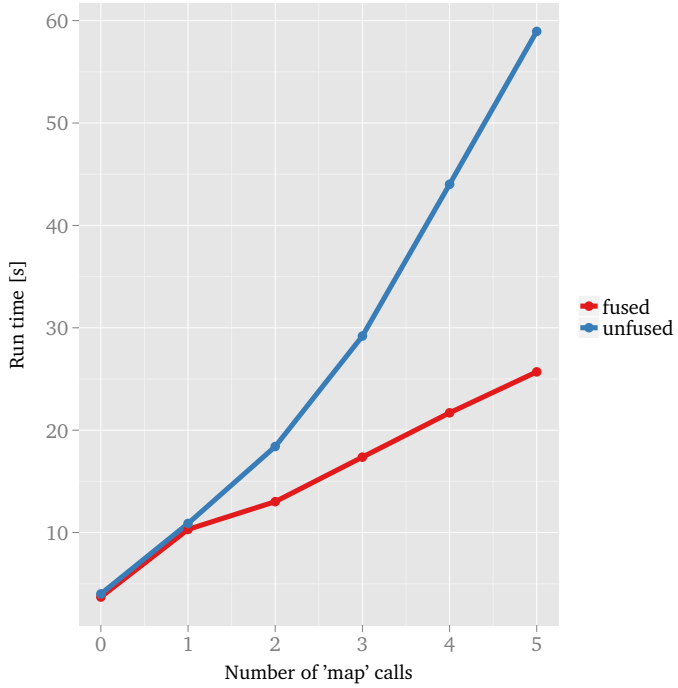


Figure 5.3: Runtime performance of fused mapping

5.4 Parallel propagation & Dead code elimination

5.4.1 Flat Fibonacci tree

Our first test case is a program that calculates Fibonacci numbers inside of a reactive tree-like structure. The root of the tree is a mutable reactive that holds the integer value that gives us the wanted position of the Fibonacci number to calculate. Depending on the root, we create reactives that calculate the requested Fibonacci number. A visualization of the dependency graph resulting from this program can be seen in figure 5.4.

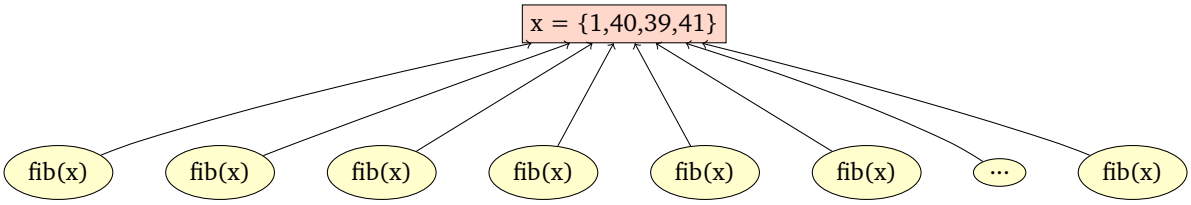


Figure 5.4: Flat fibonacci tree

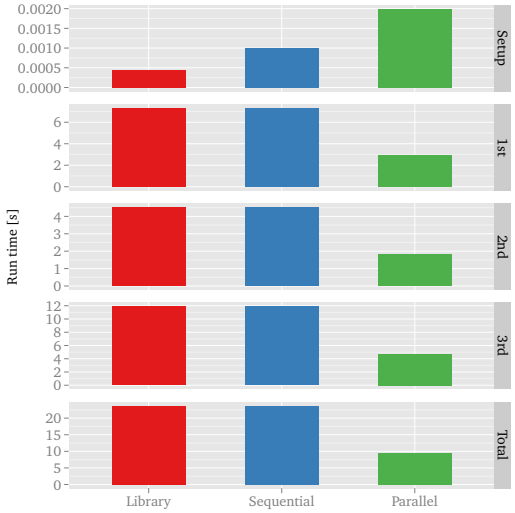
A single run of the benchmark goes through the following steps: Initially, the root of the tree is set to 1. Afterwards, the `Signals` are instantiated. This marks the first measure point, *Setup Time*. The remainder of the run changes the content of the root to a different value three times and measures the time needed to update the dependency graph (*1st propagation, 2nd propagation, 3rd propagation*). Finally, we measure the execution time of the whole run (*Total Time*).

Figure 5.5 shows the result of the benchmark and visualizes the numbers. SimpleReactive as a standalone library (Library) has comparable run times to OptiReactive using the propagation algorithm of SimpleReactive (Sequential). OptiReactive with parallel propagation enabled (Parallel) is about twice as fast as the other two versions. This result shows that reactive programs can benefit dramatically from parallel propagation. As expected, both OptiReactive versions exhibit a marginally higher warmup time due to the form of the generated code, which is asymptotically negligible.

If we use the same program as in the previous benchmark, but only reference (i.e. by printing to the console or using it in another expression) ten out of the twenty `Signals`, the built-in dead code elimination will remove the unreferenced half, leading to even better runtime performance. The results of the described benchmark are shown in figure 5.6. We can see that both OptiReactive versions run about twice as fast as in figure 5.5, because only half as much Fibonacci

	Library	Sequential	Parallel
Setup time	0.4545455	1.0000000	2.0000000
1st propagation	7333.909	7328.091	2914.545
2nd propagation	4522.000	4530.273	1807.273
3rd propagation	11851.182	11898.727	4733.909
Total time	23707.545	23758.091	9457.727

(a) Run time in ms



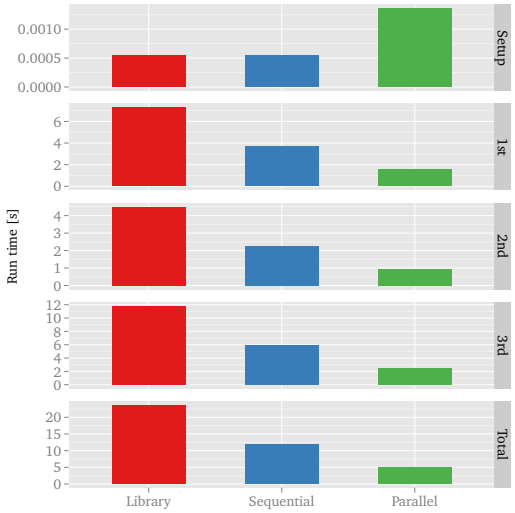
(b) Run time visualized

Figure 5.5: Results for the flat Fibonacci tree

computations have to be done during one propagation. The same program written in SimpleReactive does not show any improvement because there is no dead code elimination for reactives when using pure Scala.

	Library	Sequential	Parallel
Setup time	0.5454545	0.5454545	1.3636364
1st propagation	7300.273	3667.909	1553.545
2nd propagation	4513.5455	2258.7273	947.7273
3rd propagation	11819.909	5920.182	2459.909
Total time	23634.273	11847.364	4962.545

(a) Run time in ms



(b) Run time visualized

Figure 5.6: Results for the flat fibonacci tree with less references

5.4.2 Deep Fibonacci tree

As a second test case, we use a tree of Fibonacci computations, with the dependency graph structure shown in figure 5.7. Note that the dependencies between children and the root are omitted. The execution of one run follows the same methodology as in the previous benchmark in section 5.4.1.

The results of the regular execution are shown in figure 5.8. Unsurprisingly, the running times of all three versions are almost the same because the parallel propagation only speeds up the propagation from the root node to the first level of children and not further. Although not by much, OptiReactive with parallel propagation performs better than the sequential version which itself is slightly better than the pure library version using SimpleReactive.

If we keep the same graph structure, but reduce the number of actually referenced reactives, dead code elimination can speed up the program. This circumstance can be easily demonstrated by a variant of the previous experiment. In the

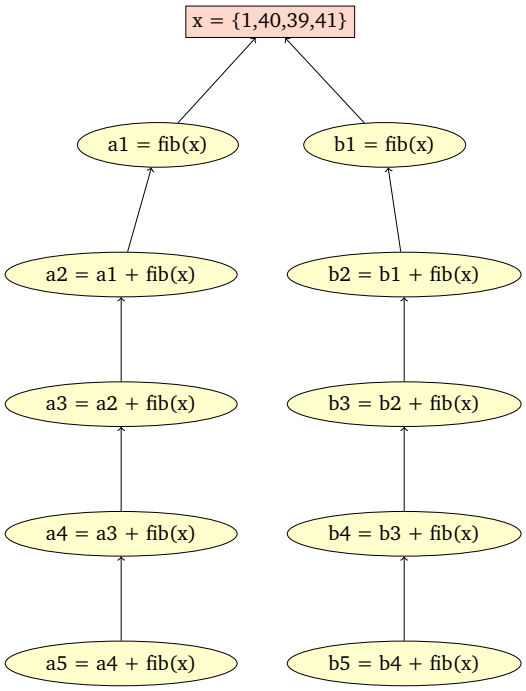
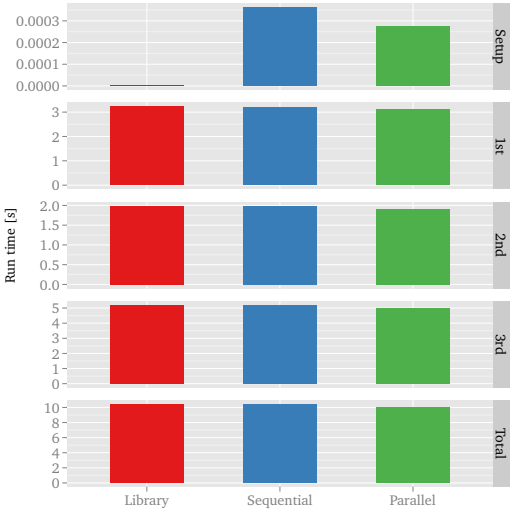


Figure 5.7: Deep Fibonacci tree

	Library	Sequential	Parallel
Setup time	0.0000000	0.3636364	0.2727273
1st propagation	3237.000	3200.000	3110.636
2nd propagation	1995.091	1981.455	1908.273
3rd propagation	5210.818	5172.273	5006.818
Total time	10442.91	10354.09	10026.00

(a) Run time in ms



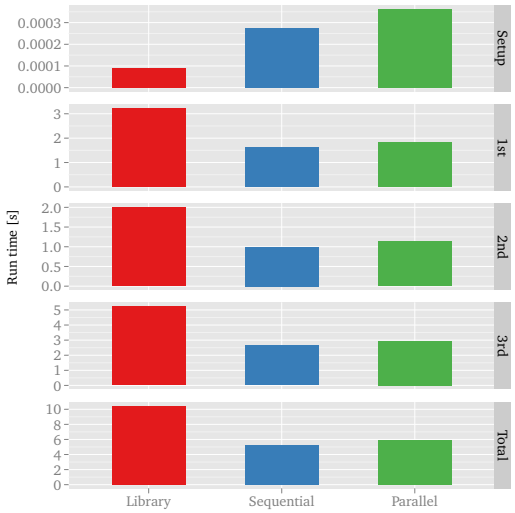
(b) Run time visualized

Figure 5.8: Results for the deep Fibonacci tree

variation the root node references only one of the two branches, The results for the case are shown in figure 5.9. Again on account of dead code elimination, we see that both OptiReactive versions run twice as fast as in figure 5.8, while the version written in SimpleReactive shows no improvement.

	Library	Sequential	Parallel
Setup time	0.09090909	0.27272727	0.36363636
1st propagation	3233.636	1632.000	1815.091
2nd propagation	2000.3636	994.9091	1126.6364
3rd propagation	5222.364	2639.636	2945.000
Total time	10456.455	5266.818	5887.091

(a) Run time in ms



(b) Run time visualized

Figure 5.9: Results for the deep Fibonacci tree with less references

We note that while we referenced only the `a5` Signal, resulting in the complete removal of the right path, any Signal that is not referenced will speed up the execution, albeit not as dramatically as in the previous case, where one whole path is eliminated.

5.5 Usability

Using the DSL. The syntax of OptiReactive is a super set of SimpleReactive, which makes it easy to convert programs between the two. When using syntactic transformations, the syntax of OptiReactive programs is shorter and porting them to SimpleReactive may require some additional work. When using Delite, the users of the DSL are provided with a compilation and execution infrastructure that makes it easier to use the programs written in the DSL. Using only LMS allows for more dynamic compilation and execution during regular programs, but the user has to take care of compiliation and excecution.

Developing optimizations. While the actual process of developing individual optimizations is straight forward, it requires knowledge of multistage programming in general as well as some specifics of the used library, LMS. The extensive use of traits in the framework allows composition of individual components in a flexible way, but is hardly traceable at the beginning.

External libraries. A big drawback of using OptiReactive and LMS in general is, that it is not possible to mix arbitrary external library code with DSL code. The lifting of the operations has to be done manually by the DSL author which can be quite cumbersome if there are a lot. By default, LMS comes with traits that do some of the lifting for many standard Scala operations, but the support is by far not complete and lacks some essential methods, for example on collections. Some recent developments start to address this issue, for example *Scopes*, an addition to the Scala-Virtualized compiler. A Scope is a “compiler-provided type that acts as a tag to encapsulate DSL blocks” [SRB+13], and enables composition of DSL and arbitrary host code blocks.

Tool chain. LMS requires a modification of the standard Scala compiler, provided by Scala-Virtualized. While LMS can be used as a prepackaged library, at the time of writing this is not the case for Delite. Getting the setup correct is not easy because of interdependencies between the used LMS and Delite versions. Delite however is currently in alpha only and under active development so it is to be expected that the user experience will improve, given some time.

6 Related Work

In this chapter we reference the research that is strictly related to the work presented in this thesis. An exhaustive overview is out of the scope of this chapter. The reader can refer to [SHM14] for a survey on reactive programming and to [SM13] for the connections with related fields and open research issues.

6.1 Reactive programming in Scala

6.1.1 Scala.React

Scala.React [MRO10] integrates composable reactive programming abstractions into Scala. It has a discrete model of time in the sense that the propagation proceeds in cycles. A propagation cycle is divided into two phases: first, all reactives are updated, and second, all observers are run. The reason for this is to avoid inconsistent data, called *glitches* that might be seen by observers. The propagation model itself is purely push-driven. To further ensure that inconsistencies are avoided, the dependency graph is topological sorted. During a propagation cycle all reactives are updated in their topological order. Dynamic dependencies are handled by throwing exceptions and reevaluation of the respective entities.

6.1.2 REScala

REScala unifies abstractions of reactive languages and object-oriented programming and "[...] supports the development of reactive applications by fostering a functional declarative style which complements the advantages of object-oriented design" [SHM14]. It extends the design of EScala [GSM+11] and unifies imperative events and reactive values, thereby supporting a mixed object-oriented and functional style. Additionally, REScala comes with a rich API to combine events and reactive abstractions, which makes it easy to compose the best of both worlds.

6.2 LMS and Delite

6.2.1 Scala-Virtualized

Scala-Virtualized [MRHO12] is a customized version of the standard Scala compiler and library, that provides enhanced support for developing DSLs in Scala. It allows the definition of infix extension methods on existing types and overloads Scala's control structures, variables, object creation etc. In addition Scala-Virtualized provides implicit source locations which reify source information and if added to the method signature are generated by the Scala compiler without further actions.

6.2.2 Lightweight Modular Staging

Lightweight Modular Staging (LMS) [RO10] is a dynamic code generation approach, provided in form of a library in Scala, that uses staging [JS86]. LMS is related to multi-stage programming (MSP) [Tah99]. It differs from other MSP as Mint [WRI+10], MetaOCaml [CTHL03] and MetaML [TS00] which use a syntactic MSP approach. In LMS binding times are distinguished using only types.

6.2.3 Delite DSLs

OptiML [SLB+11] is a domain specific language for machine learning using Delite [CSB+11]. The evaluation of different machine learning algorithms indicates that code written using OptiML can outperform optimized MATLAB code in most cases. Machine Learning algorithms are a good use case because most of them have large parts of data- and calculation-parallelism that can be exploited on "heterogeneous hardware" [AMD08]. The main benefit of using OptiML instead of MATLAB is that the Delite compiler generates efficient, domain-specific optimized code for heterogeneous devices

without exposing parallelism or device details to users of the DSL. **OptiQL**, heavily inspired by LINQ [MBB06] is "a DSL for data querying of in-memory collections" [Rom12]. **OptiMesh** implements Liszt [DJP+11], a "DSL for mesh-based partial differential equation solvers" [Rom12]. **OptiGraph** is a DSL based on Green-Marl [HCSO12], a DSL for static graph analysis. **OptiCollections** [Rom12] provides collections that are optimized via the optimizing compilation techniques of Delite. The optimized version can be used transparently in DSLs and speedup common operations.

6.3 Optimizations for reactive programming

6.3.1 FrTime - Lowering

The idea of **Lowering** [BCK07] is similar to our "Constant folding" optimization. The technique was developed to improve the performance of FrTime [CK06], an FRP approach in DrScheme [FCF+02]. FrTime, unlike our reactive libraries, uses *implicit* lifting of functions to construct the dataflow graph. The graph constructions does not happen statically, as it is the case in our work, but dynamically, allowing incremental development. In fact, the optimized result of figure 4.5, shown in listing 4.3 is similar to an example shown in [BCK07], Figure 5. A main difference is that the lowering approach makes no static dataflow analysis, something that is natural using our DSL and LMS.

6.3.2 Causal Commutative Arrows

Causal Commutative Arrows (CCA) [LCH11] are an optimization technique that is applicable to a subclass of the `ArrowLoop` typeclass for Arrows [Hug00], named `ArrowInit`. Additionally, instances should satisfy a product and commutativity law. For CCAs the authors provide a normal form together with a normalization procedure. Finally, an optimization is presented, that is based on the observation that "any CCA program can be transformed into a single loop containing one pure arrow and one initial state value" [LCH11]. The optimization is implemented via Template Haskell [SJ02] and is therefore performed at compile time. Essentially, recursive arrows are transformed into a single imperative loop, removing much of the typical overhead of arrow loops. The static compilation restricts the applicability, as only information known at compile time can be used. Dynamic reactivity (higher-order reactivity) can therefore not be optimized via this technique.

6.3.3 Dynamic Optimization for FRP using GADTs

Yampa [NCP02] is a domain specific language for functional reactive programming in the form of an arrow combinator library. Using generalized algebraic data types (GADTs) in Haskell [JWW04, JVWW06], [Nil05] proposes an optimization that further optimizes Yampa. Yampa already uses some optimizations, but, according to the authors, many more could not be implemented because of the restricted Haskell98 type system. The authors of [Nil05] describe general arrow optimizations together with some Yampa specific ones. For example, Yampa's event processing is optimized by using GADTs to enable the handling of event-processing functions specially. This approach is similar to what we do via LMS, as we are using IR nodes for operations on which we can match and eventually optimize by changing the representation.

7 Conclusion & Future work

In this work, we presented OptiReactive, a domain-specific language for reactive programming based on multistage programming. In addition, we introduced SimpleReactive, a reactive library which is used as the target of the DSL generated code. In this thesis, we explored several optimizations provided by the synergies of multistage and reactive programming. The evaluation of OptiReactive shows promising results regarding runtime performance improvements of each optimizations. Besides performance optimizations, syntactic transformations like dependency inference can be performend at staging time. As a result, they have no negative impact on the programs' performance, while still enriching the base library with additional constructs or more concise syntax.

7.1 Research Perspectives

The design space of possible optimizations emerging from the combination of multistage and reactive programming is largely unexplored. In future work, besides new optimizations, we see at least two main areas that remain open to further development. Firstly, in future versions of Delite the issues regarding parallel propagation may be solved. As the benchmarks showed, parallel propagation can drametically improve the performance of a reactive program. Therefore we recommend further exploring this area. Secondly, we observe that when reactivity is modeled using a dependency graph, transformations are very easy to perform. The constant folding optimizations could be extended to collapse unreferenced intermediate reactives in paths of the dependency graph, thereby reducing the total number of updates that have to be done.

Bibliography

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2nd edition, September 2006.

[AMD08] AMD. The industry-changing impact of accelerated computing. White paper, 2008.

[BCK07] Kimberley Burchett, Gregory H. Cooper, and Shriram Krishnamurthi. Lowering: a static optimization technique for transparent functional reactivity. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '07, pages 71–80, New York, NY, USA, 2007. ACM.

[CK06] Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Proceedings of the 15th European conference on Programming Languages and Systems*, ESOP'06, pages 294–308, Berlin, Heidelberg, 2006. Springer-Verlag.

[CSB+11] Hassan Chafi, Arvind K Sujeeth, Kevin J Brown, HyoukJoong Lee, Anand R Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pages 35–46. ACM, 2011.

[CTHL03] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using asts, gensym, and reflection. In *Generative Programming and Component Engineering*, pages 57–76. Springer, 2003.

[DJP+11] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. Liszt: a domain specific language for building portable mesh-based pde solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12, New York, NY, USA, 2011. ACM.

[EH97] Conal Elliott and Paul Hudak. Functional reactive animation. *SIGPLAN Not.*, 32(8):263–273, August 1997.

[FCF+02] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. Drscheme: a programming environment for scheme. *J. Funct. Program.*, 12(2):159–182, March 2002.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 11 1994.

[GSM+11] Vaidas Gasiunas, Lucas Satabin, Mira Mezini, Angel Núñez, and Jacques Noyé. Escala: modular event-driven object interactions in scala. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 227–240, New York, NY, USA, 2011. ACM.

[HCSO12] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-marl: a dsl for easy and efficient graph analysis. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 349–362, New York, NY, USA, 2012. ACM.

[Hug00] John Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111, May 2000.

[JS86] Ulrik Jørring and William L Scherlis. Compilers and staging transformations. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 86–96. ACM, 1986.

[JVWW06] Simon P. Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 50–61, New York, NY, USA, 2006. ACM.

[JWW04] P. Jones, S. Washburn, and G. Weirich. Wobbly types: Type inference for generalised algebraic data types, 2004.

[LCH11] Hai Liu, Eric Cheng, and Paul Hudak. Causal commutative arrows. *J. Funct. Program.*, 21(4-5):467–496, September 2011.

[MBB06] Erik Meijer, Brian Beckman, and Gavin Bierman. Linq: reconciling object, relations and xml in the .net framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 706–706, New York, NY, USA, 2006. ACM.

[MGB+09] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: a programming language for ajax applications. *SIGPLAN Not.*, 44(10):1–20, October 2009.

[MPO08] Adriaan Moors, Frank Piessens, and Martin Odersky. Generics of a higher kind. *SIGPLAN Not.*, 43(10):423–438, October 2008.

[MRHO12] Adriaan Moors, Tiark Rompf, Philipp Haller, and Martin Odersky. Scala-virtualized. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, pages 117–120. ACM, 2012.

[MRO10] Ingo Maier, Tiark Rompf, and Martin Odersky. Deprecating the observer pattern. *Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, Tech. Rep*, 2010.

[NCP02] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, Haskell '02, pages 51–64, New York, NY, USA, 2002. ACM.

[Nil05] Henrik Nilsson. Dynamic optimization for functional reactive programming using generalized algebraic data types. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 54–65, New York, NY, USA, 2005. ACM Press.

[OAC+04] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, Citeseer, 2004.

[OZ05] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 41–57, New York, NY, USA, 2005. ACM.

[RO10] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *SIGPLAN Not.*, 46(2):127–136, October 2010.

[Rom12] Tiark Rompf. *Lightweight Modular Staging and Embedded Compilers: Abstraction Without Regret for High-Level High-Performance Programming*. PhD thesis, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, 2012.

[RSL+11] Tiark Rompf, Arvind K Sujeeth, HyoukJoong Lee, Kevin J Brown, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Building-blocks for performance oriented dsls. *arXiv preprint arXiv:1109.0778*, 2011. - trait hierarchy on p.4 - description of scheduling and kernel calls on p.8 - regular compiler optimizations on p.9 - effect system issues p.10 - advantages of doing staging time calculations, even if there are dynamic dependencies - Related work: dsls + multi stage.

[SHM14] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. REScala: Bridging between object-oriented and functional style in reactive applications. *MODULARITY AOSD*, 2014.

[SJ02] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, Haskell '02, pages 1–16, New York, NY, USA, 2002. ACM.

[SLB+11] Arvind K Sujeeth, HyoukJoong Lee, Kevin J Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand R Atreya, Martin Odersky, and Kunle Olukotun. Optiml: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning, ICML*, 2011.

[SM13] Guido Salvaneschi and Mira Mezini. Reactive behavior in object-oriented applications: an analysis and a research roadmap. In *Proceedings of the 12th annual international conference on Aspect-oriented software development*, AOSD '13, pages 37–48, New York, NY, USA, 2013. ACM.

[SRB+13] ArvindK. Sujeeth, Tiark Rompf, KevinJ. Brown, HyoukJoong Lee, Hassan Chafi, Victoria Popic, Michael Wu, Aleksandar Prokopec, Vojin Jovanovic, Martin Odersky, and Kunle Olukotun. Composition and reuse with compiled domain-specific languages. In Giuseppe Castagna, editor, *ECOOP 2013 – Object-Oriented Programming*, volume 7920 of *Lecture Notes in Computer Science*, pages 52–78. Springer Berlin Heidelberg, 2013.

[Tah99] Walid Taha. *Multi-stage programming: Its theory and applications*. PhD thesis, Citeseer, 1999.

[TS00] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, October 2000.

[WRI+10] Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. Mint: Java multi-stage programming using weak separability. *ACM Sigplan Notices*, 45(6):400–411, 2010.