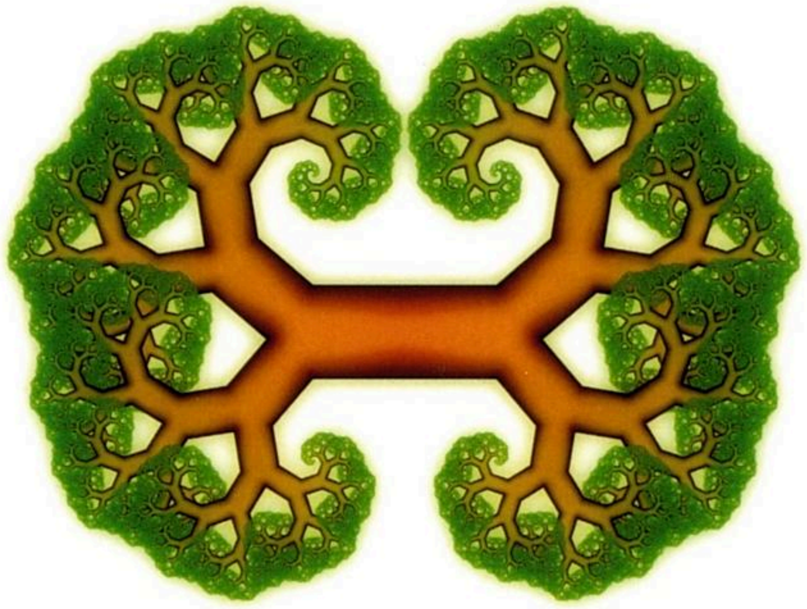
Efficient Abstractions for Visualization and Interaction



Atze van der Ploeg

Efficient Abstractions for Visualization and Interaction

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. D.C. van den Boom
ten overstaan van een door het college voor promoties ingestelde
commisie, in het openbaar te verdedigen in de Agnietenkapel
op woensdag 8 april 2015, te 14:00 uur
door

Atze Johannes van der Ploeg

geboren te Amsterdam

The work in this thesis has been carried out at Centrum Wiskunde & Informatica (CWI) in Amsterdam, under the auspices of the research school IPA (Institute for Programming research and Algoritmics).

Acknowledgements

I may not have gone where I intended to go, but I think I have ended up where I intended to be.
– Dirk Gently in *The Long Dark Tea-Time of the Soul* by Douglas Adams

It has been an eventful four years for me, filled with sorrow and euphoria, dead ends and achievements, moments of confusion and moments of clarity, struggles and acceptance, and yes, even a dash of romance, making it a proper adventure. I have grown considerably, both intellectually and as a person. In such a way even, that I now cannot imagine the naivety I had just four years ago. Of course, I'm still the loud, opinionated, blunt Dutchman I've always been, but I do believe my world views are now a bit more realistic, and that my strong opinions are now slightly more weakly held. Reaching the end of this journey, some thanks are in order.

My promotor, Paul Klint, I've been told that I am but one in a long line of stubborn PhD students, but I dare to theorize that I am among the most stubborn. Your liberal style of supervision has made me an independent researcher and more critical thinker. I thank you for your constant support and our stimulating conversations, which were always enjoyable, whether we were discussing software engineering or sharing astonishment on the way the world works. Tijs van der Storm, I've learned a lot from you, on software engineering, doing research and the politics of science. I'll miss our daily stream of "peukie-tijd" moments and your never-ending quest to find the worst bar in town.

Jan van Eijck, I have very much enjoyed your company and our discussions on functional programming, logic, philosophy and life. I regret that we never worked together, and hope we will do so in the future, which would be, I conjecture, a pleasant experience. Davy Landman, I enjoyed your company, and I'll miss your ability to, at times, be even more blunt than me. Ali Afroozeh, I enjoyed our shared venting of frustrations, through conversation as well as heavy metal music. Riemer van Rozen, never have I met such an easy-going friendly giant. Your stories about the workings of the HvA have made me less naive and reminded me how privileged my position actually was. My other colleagues, Floor Sietsma, Pablo Inostroza Valdera, Jouke Stoel, Bas Basten, Jurgen Vinju, Robert van Liere, Ashim Shahi, Anastasia Izmaylova, Bert Lisser and Vadim Zaytsev, you're company has been a joy to experience and I thank you for putting up with the loud conversation I periodically bellowed into your offices from rooms away.

Jurriaan Rot, I really like your extremely critical, anti-pretentious views on math, science and the world and our anti-pretentious discussions over beer on pretentious subjects have been very memorable for me. Tom Sterkenburg, I'll miss our meetings which consisted of a nice mix of nonsense, philosophy, math and other discussions and, perhaps most importantly, coffee. Pieter Hijma, I admire your persistence to keep pursuing your programming vision and am happy that this road has eventually been fruitful. We've been keeping in touch since our time at the VU, and I look forward to extending this relation into future years. Harry Buhrman, for being the friendly face from the the other side of the hall, and for giving me heartfelt advice when I was in superposition on going to Sweden. Koen Claessen, talking to you on functional programming was a breath of fresh air, and your enthusiasm has rekindled my interest in a scientific career. I very much look forward to working with you at Chalmers.

My family, Menno, Jelmer, Aniek, Tieke and Tymen, thank you for your support and warmth. My friends Dineke, Patrick, Willem, Olivier, Mels, Maurice, Worf, Michal, Sander, Daniël, Mark, Irma, Les, Bas, Stef, Jurjen, Hieke, Thomas, Jadwiga, thanks for your company, conversation and parties which always gave me a way to escape from science. My cat, Frisbee, thank you for your cuddles and never-ceasing purring, and for always seeming to think that my ideas are out of this world. My love, Jonna, thank you for your love, warmth and support, even during my unreasonable rants.

Contents

1

Introduction

Abstractions, in the form of functions, methods and classes, are an essential tool for any programmer. Abstractions encapsulate the details of a computation, so that the programmer only needs be aware of what an abstraction achieves, and not how it achieves it. Programmers compose invocations of abstractions to obtain the behavior they want. However, sometimes compositions come at a cost, such as:

- The resulting program is too slow.
- The program takes up too much memory.
- The behavior of the program is hard to understand.
- The time/memory requirements of the program are hard to predict.
- The behavior of the program becomes non-deterministic because the abstractions involve concurrency.
- The (numerical) precision of the program is not high enough.

As a result the programmers may be forced to not use some abstractions, instead creating specialized versions of them by hand.

In this thesis we aim to make this situation less likely in the domain of interactive visualizations. The abstractions for programming interactive visualizations fall broadly into three categories, namely:

- Graphics abstractions (draw a line, rotate, fill, ...).
- Layout abstractions (tree layout, force directed graph layout, ...).
- Event/time abstractions (register listener, wait for an event, ...).

This thesis presents novel efficient abstractions in each of these areas. More specifically, we contribute the following:

- Abstractions for graphics that allow us to discretize later, thereby preventing sampling artifacts and making non-affine transforms more efficient.
- A tree layout algorithm that produces a layout in linear time, whereas previous methods for the same kind of layout either cost quadratic time or produce less compact layouts.
- An efficient evaluation mechanism for Functional Reactive Programming, a programming framework which provides abstractions for dealing with time and events.
- A technique to increase the performance of series of associative operators from quadratic to linear for a class of associative operators, which is, among others, applicable to Functional Reactive Programming.

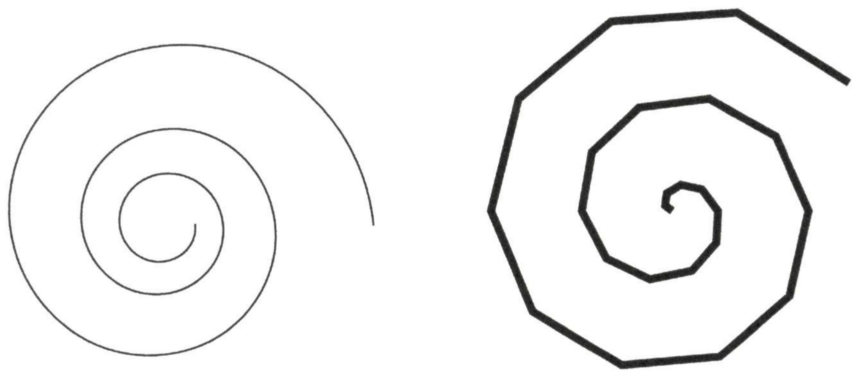
The work in this thesis was motivated by our efforts on the Figure library [Klint et al., 2011] in the Rascal[1] language for software analysis and transformation. This framework intends to offer composable abstractions for the interactive visualizations of software artifacts. This thesis presents novel results for the abstractions used when programming interactive visualizations, namely programming graphics, layout algorithms and programming interactive systems. These results create a more versatile foundation for higher level frameworks such as the Rascal Figure library. In the following subsections we briefly explain the problems and introduce our results in each of these areas.

1.1 Graphics Abstractions

A fundamental ingredient for programming visualizations is graphics abstractions: abstractions such as drawing a line, filling a shape or rotating a shape. This is done through frameworks such as Processing[2], DirectX[3] or OpenGL[4]. These frameworks provide a set of shape primitives, textures and transformations to compose. However, if what we want to express is not directly expressible as a combination of such shape, texture and transformation primitives, then we have to approximate it using these primitives. When we then again want to compose that approximation with transformations such as scaling up, artifacts due to the approximation will become visible. To make this more concrete, let us introduce three examples of this.

An example of a shape that is not directly supported by the typical shape primitives in these frameworks is the spiral of Archimedes[5], shown in Figure 1.1(a). Traditional graphical frameworks, such as DirectX, Processing and OpenGL, only offer a limited set of shape primitives, most commonly Bézier curves[6] up to cubic order

[1]http://www.rascalmpl.org
[2]https://www.processing.org/
[3]http://msdn.microsoft.com/library/directx
[4]http://www.opengl.org/
[5]http://mathworld.wolfram.com/ArchimedesSpiral.html
[6]http://mathworld.wolfram.com/BezierCurve.html



(a) Archimedes' Spiral.

(b) A scaled up approximation of Archimedes' Spiral.

Figure 1.1: Versions of Archimedes' Spiral.

(straight lines are also Bézier curves). However, the spiral of Archimedes is not a collection of Bézier curves. It can be approximated with Bézier curves, but a good approximation is far from obvious and requires quite some knowledge of mathematics[7]. An approximation using straight lines is easier than an approximation using quadratic or cubic Bézier curves, but is also less precise. Most importantly, an approximation is not resolution independent: the description of the spiral depends on the resolution at which it is drawn. Unless measures are taken, we will see the jaggedness of the approximation if the drawing is scaled up, as shown in Figure 1.1(b).

Traditional graphics frameworks offer a set of textures, namely color fillings, linear gradients, radial gradients and images. An example of a texture that is not in this set, is the cushion texture used to fill the rectangles in cushion treemaps [Van Wijk and Van de Wetering, 1999], as shown in Figure 1.2. Again, one can approximate this texture, by specifying the pixels of a discretization of the texture, but this approach is not resolution independent. We then cannot freely compose with transformations without losing quality: rendering artifact will be visible as show in Figure 1.3.

As a third example, suppose we have programmed a visualization using a traditional graphics framework and we now want to add a focus+context lens [Carpendale and Montagnese, 2001], such as the one shown in Figure 1.4. Since only affine transformations[8] (transformations that preserve straight lines) are supported, we can only support such a transformation by an approximation. One technique that has been proposed [Pietriga et al., 2010] is to render whatever is under the lens twice: once without magnification and once with magnification. Afterwards, both renderings are

[7] `http://math.stackexchange.com/questions/179000`

[8] `http://mathworld.wolfram.com/AffineTransformation.html`

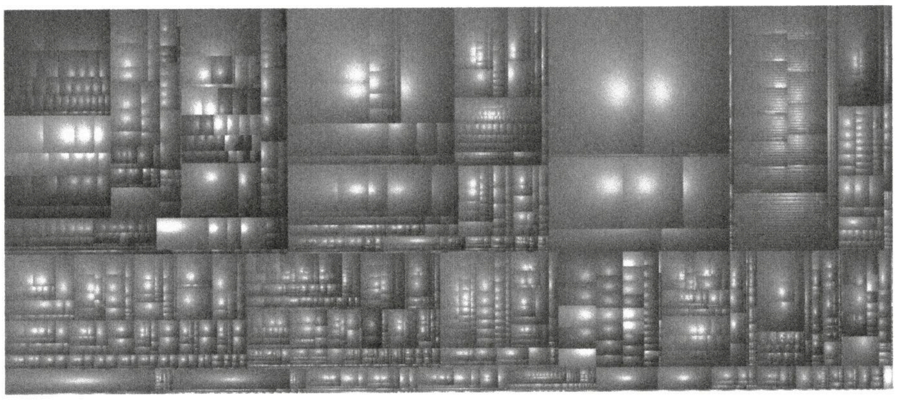
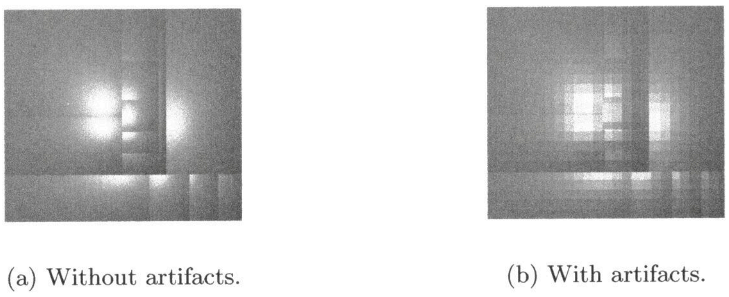


Figure 1.2: Cushion Treemaps [Van Wijk and Van de Wetering, 1999].


(a) Without artifacts.

(b) With artifacts.

Figure 1.3: Small cushion treemap without and with artifacts.

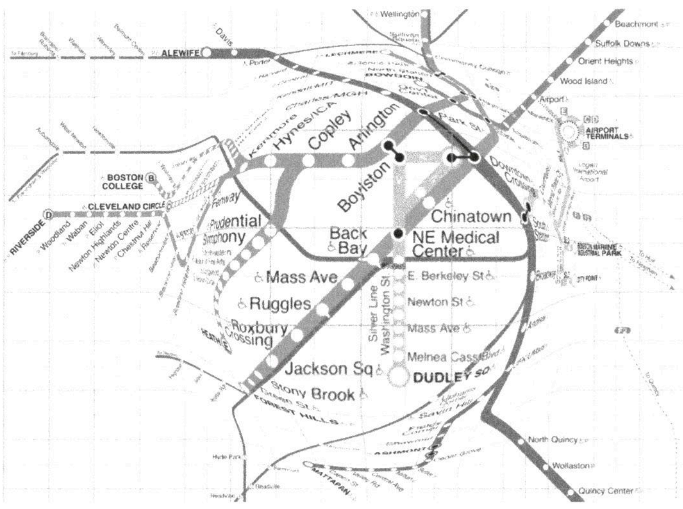


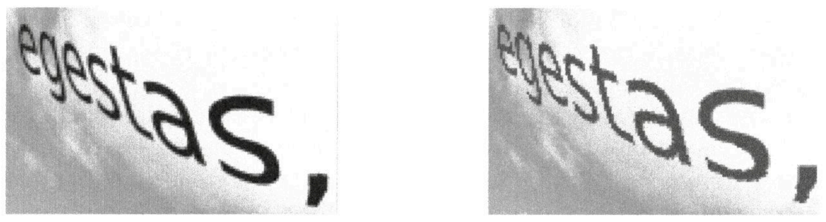
Figure 1.4: Focus+context lens (image taken from [Pietriga et al., 2010]).

combined to produce the lens area. The second, magnified rendering uses a buffer of width and height proportional to the zoom factor, making higher zoom factors quadratically more expensive. Moreover, combining two non-transformed drawings to produce a transformation can lead to rendering artifacts, as shown in Figure 1.5. Also, as with the previous approximations, if we afterwards want to scale up the lens area, we have to take measures to prevent artifacts.

This brings us to our first research question:

Research Question 1 *Is it possible to program 2D graphics in a declarative way that is general, simple, expressive, composable and resolution-independent while still being efficient?*

In Chapter 2, we answer this question with a "yes" by presenting a library for



(a) Without artifacts.

(b) With artifacts.

Figure 1.5: Detail of a focus+context lens with and without artifacts due to sampling (image taken from Chapter 2).

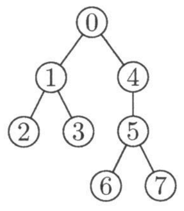


Figure 1.6: Example of a tree layout produced by the Reingold-Tilford algorithm.

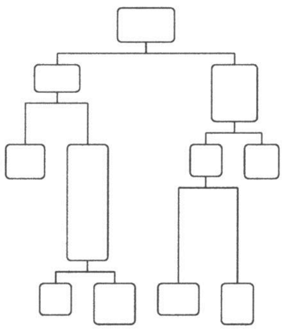
declarative resolution-independent 2D graphics. This library generalizes and simplifies the functionality of traditional frameworks, while preserving their efficiency. As an example, we show the implementation of a focus+context lenses that gives better image quality and better performance than a solution using a traditional graphics framework at a fraction of the code.

1.2 Layout Algorithms

Layout algorithms are algorithms which make the structure of the input data concrete by presenting it spatially. Research into such algorithms has yielded a host of algorithms, such as algorithms for visualizing graphs [Von Landesberger et al., 2011], trees [Reingold and Tilford, 1981; Walker, 1990; Johnson and Shneiderman, 1991; Van Wijk and Van de Wetering, 1999], and enforcing spatial constraints [Borning et al., 1997].

An example of a layout algorithm is the Reingold-Tilford algorithm, which produces drawings of trees such as the one shown in Figure 1.6. Sometimes, we also want to show some information inside each node or in the dimensions of each node. Examples of this are Tableau style proof trees, parse trees of (formal) languages, class diagrams in software engineering and Polymetric views [Lanza and Ducasse, 2003a] . The latter are inheritance diagrams of software systems where the width and height of each node signifies a software metric of the corresponding class, such as the lines of code or number of methods. In these situations, the width and height of each node may vary. The Reingold-Tilford algorithm will then produce a layered drawing, as show in Figure 1.7(a). In such a layered drawing, all nodes at the same depth in the trees are given the same top vertical position.

However, such layered drawings may then use more vertical space than necessary. A non-layered drawing of a tree places children at a fixed distance from the parent, thereby giving a more vertically compact drawing. The difference between a layered and a non-layered drawing is shown in Figure 1.7. Non-layered drawings can also be used to draw trees where the vertical position of each node is given. An example of this is a family tree diagram where the vertical top coordinate of a node signifies the birth year of the corresponding person, as shown in Figure 1.8. An example in biology is a diagram which shows evolutionary relationships between biological species and the time in which each species came into existence.


(a) Layered.

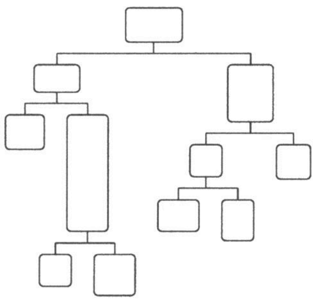

(b) Non-layered.

Figure 1.7: Layered and non-layered drawings of the same tree.

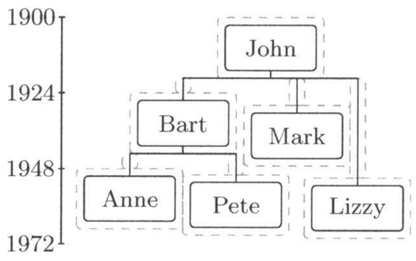


Figure 1.8: Descendants of John: layout with prescribed vertical positions (corresponding to birth year).

While a layered tree layout is produced in linear time using the Reingold-Tilford algorithm [Reingold and Tilford, 1981], the fastest known algorithm for non-layered trees runs in quadratic time [Bloesch, 1993]. This meant that programmers had to make a trade-off between speed and a compact layout, or create a specialized algorithm for her particular needs. This leads to the second question:

Research Question 2 *Is it possible to produce non-layered layouts of trees in linear time?*

We present a linear time algorithm for producing non-layered tree layouts in Chapter 4. More precisely, our algorithm is a modification of the Reingold-Tilford algorithm, but the original complexity proof of the Reingold-Tilford algorithm uses an invariant that does not hold for the non-layered case. We give an alternative proof of the algorithm and its extension to non-layered drawings. To improve drawings of trees of unbounded degree, extensions to the Reingold-Tilford algorithm have been proposed. These extensions also work in the non-layered case, but we show that they then cause a $O(n^2)$ run-time. We present a modification to these extensions that restores the $O(n)$ run-time.

1.3 Programming Interactive Systems

Interactive visualizations are *reactive*: they engage in a dialogue with their environment, reacting to events as they arrive. For example, interactive visualizations need to respond to mouse movements, key presses, network messages or touch commands. Programming such interactivity is done using abstractions that allow us to react to events. Abstractions to react to events, allow us to do one of the following:

- Wait for an event to happen.
- Give instructions on what to do when an event happens.

Waiting for an event to happen, also known as blocking I/O, means that the program invokes a method which has the effect that the entire program is suspended (blocked) until the desired event, such as a mouse click, occurs. This model has obvious drawbacks for composability: what if we want to compose two program components in parallel which both can wait for an event? The standard composition tools of the programmer, like forming expressions and sequencing statements, do not support this.

There are two ways to deal with this:

- Give up on composing reactive programs and instead organize the program as a monolithic event loop. In this approach, a main loop repeatedly gathers the events that should be waited for and performs the actions, such as drawing to the screen or sending network messages, that should be done. At the end of each iteration, the program invokes a method that waits until one of the requested events occurs, by use of, for example, the Unix `select` method[9]. This approach

[9]`http://pubs.opengroup.org/onlinepubs/007908799/xsh/select.html`

is simple, but is not entirely satisfactory: the programmer must manually route which events to wait for and the results of such events to the desired components, leading to boilerplate code.

- Use concurrency, running each component on a separate thread or process. The communication between components is then done via shared data with locking or via message passing. However, concurrency is non-deterministic: the actions of the program not only depend on which events occurred and when, but also on the interleaving of threads. This can make the behavior of the program hard to predict and understand. When message passing is used, it is however still possible to reason formally about such systems to and prove that the program is well-behaved using process calculi such as the Algebra of communicating processes [Bergstra and Klop, 1985] or Communicating sequential processes [Brookes et al., 1984].

The second way to deal with events, known as non-blocking I/O, is to give instructions on what to do when an event occurs. In this model, the program installs some handler code, known as the callback method, that should be run when a certain event occurs and then immediately continues, without waiting for the event to occur. An example of this is shown in Figure 1.9, which lists a Java program which calls a method doDoubleClick() when it detects a double click (two successive clicks within 0.2 seconds).

An unfortunate drawback however, it that this approach leads to the control in the program being switched between callbacks, effectively jumping around in an unstructured way: each interleaving of callback invocations is a possibility. This is known as *inversion of control*: the control in the program is not dictated by the sequencing specified by the programmer, but by the events that occur.

When callback invocations need to communicate, they can do so via shared mutable state. In the example in Figure 1.9, invocations of handleTime and handleClick communicate whether the first click has already occurred via the afterFirstClick variable. This has drawbacks: shared mutable state is combined with unstructured control, which can be hard to understand. Moreover, this technique also means we must manually register and unregister callbacks from events, as we can also see in the example. This can make it hard to predict in which order the callbacks are processed and can lead to callback loops. We refer the reader to [Maier and Odersky, 2012] for detailed criticism on this pattern.

An alternative that is composable, does not lead to inversion of control and does not introduce unnecessary non-determinism is Functional Reactive Programming (FRP). FRP makes programming with events more deterministic than using concurrency, since the result of a FRP program only depends on which events happened when, not on the specific interleaving of threads [Elliott and Hudak, 1997]. An example of FRP is shown in Figure 1.10, which shows a Haskell program using Monadic FRP (Chapter 4) that has the same behavior as the Java program in Figure 1.9: it calls a method doDoubleClick when a double click occurs.

In this example, waiting for a double right click (doubler) is defined as three steps, which are listed after the **do** keyword. The first step is to wait for a single right click.

```
class DoubleClick{
  static final int MaxDoubleClickInterval = 200;
  boolean afterFirstClick ;
  Timer t;

  DoubleClickListener (Window w){
     afterFirstClick  = false;
     w.addMouseClickListener(this);
  }

  void handleTime(){
     afterFirstClick  = false;
  }

  void handleClick (){
     if( afterFirstClick ) {
        afterFirstClick  = false;
        t.stop ();
        doDoubleClick(); // double click happened
     } else {
        afterFirstClick  = true;
        t = new Timer(MaxDoubleClickInterval);
        t.addTimelistener(this);
     }
  }
  void doDoubleClick() { ... }
}
```

Figure 1.9: Registering a double click using callbacks in Java.

```
doubler = do rightClick
             r <- (rightClick `before` sleep 0.2)
             if r then doDoubleClick else doubler
```

Figure 1.10: Registering a double click using Monadic FRP in Haskell.

```
data [a] = [] | a : [a]

[]      ++ r = r
(h : t) ++ r = h : t ++ r
```

Figure 1.11: Definition of lists and list concatenation in Haskell.

The second step is to wait either for a second right click or 0.2 seconds to pass. To this end we pass the description of these two events, namely rightClick and sleep 0.2 to the infix function ‘before‘. This function returns if the left argument (rightClick) occurred before the right argument (sleep 0.2), as soon as at least one of these events occurred. Afterwards, a boolean stating whether a right click occurred before 0.2 seconds passed, is returned and bound to r. Finally, in the third step we check this boolean. If it was true, then a double right click occurred and we call doDoubleClick. Otherwise, we start again from the top by recursively calling doubler.

One way to think of FRP is as a variant of blocking I/O, where we can compose two blocking elements in parallel without introducing non-determinism. For instance, the double click code in Figure 1.10 first blocks for a right click, but afterwards blocks until we know if another right click occurred within 0.2 seconds. This is done using the before function, which takes two blocking computations, and composes them in parallel without introducing non-determinism.

Another way to think of FRP is as composable event loops which can communicate through a data-flow network. Since communication in this data-flow network is synchronous, i.e., all communication in this network happens conceptually simultaneously in rounds, this model does not introduce non-determinism.

A problem with FRP is efficiency: in Classical FRP [Elliott and Hudak, 1997] the space usage of the program increases linearly in time. Arrowized FRP [Courtney and Elliott, 2001] does not have this problem, but the entire program is re-evaluated after each time-step. Consequently, values are redundantly recomputed even when inputs don’t change. Evaluation strategies that prevent such redundant re-computations are known as *incremental evaluation strategies*: they only update the parts of the program that are actually out of date. This leads to the following research question:

Research Question 3 *How can we support incremental evaluation in FRP?*

In Chapter 5, we present a novel FRP formulation called Monadic FRP that is implemented in a purely functional way while preventing redundant re-computations.

However, unrelated to incremental evaluation, Monadic FRP’s performance could still be improved. It has a performance problem that is common in functional programming. The problem is as follows: in some situations the number of steps it takes to evaluate an expression depends on the placement of the brackets (the association pattern).

For example, suppose $+\!\!+$ is an associative operator that appends two lists, as defined in Figure 1.11. In this situation, the evaluation of $(\mathsf{a} +\!\!+ \mathsf{b}) +\!\!+ \mathsf{c}$ is more expensive than the evaluation of the *equivalent* expression $\mathsf{a} +\!\!+ (\mathsf{b} +\!\!+ \mathsf{c})$. This can be seen as follows: $+\!\!+$ visits all elements of the left argument, but does not observe the right element. Hence, $\mathsf{a} +\!\!+ \mathsf{b}$ costs $|\mathsf{a}|$ steps, the length of a. In $\mathsf{a} +\!\!+ (\mathsf{b} +\!\!+ \mathsf{c})$, the left arguments of $+\!\!+$ always consist of a single variable and this hence runs in $|\mathsf{a}| + |\mathsf{b}|$ steps. In $(\mathsf{a} +\!\!+ \mathsf{b}) +\!\!+ \mathsf{c}$, the left argument of the outermost invocation of $+\!\!+$ consists of another invocation of $+\!\!+$, and hence costs $2|\mathsf{a}| + |\mathsf{b}|$ steps, since $|\mathsf{a}|$ occurs twice in a left hand side of $+\!\!+$.

If we iterate this pattern, a right-associated expression:

$$(((a_1 +\!\!+ a_2) +\!\!+ a_3) \ldots +\!\!+ a_{n-1}) +\!\!+ a_n$$

is asymptotically more expensive than the equivalent left-associated expression:

$$a_1 +\!\!+ (a_2 +\!\!+ (a_3 +\!\!+ \ldots (a_{n-1} +\!\!+ a_n)))$$

At first glance, the solution to this problem might seem easy: simply only write left-associated expressions. However, this is not compositional: we must make sure that the first argument of $+\!\!+$ can never be a result of $+\!\!+$ itself. A well known cure for this dependence on the association pattern is Continuation Passing Style [Claessen, 2004; Voigtländer, 2008], but this does not work for all usage patterns: it again imposes a penalty if we alternate between using the associative operator and pattern matching on the results of that operator.

For lists and list concatenation the solution is to use better sequence data structures, such as the ones described in [Okasaki, 1998]. With such sequence data structures, sequence concatenation is efficient no matter what the usage pattern, even when alternating between using concatenation and pattern matching on the results of such concatenations.

However, the problem does not only occur with list concatenation, but also with a host of other associative operators, such as the "do this after that" operator in Monadic FRP. The solution for lists, namely better data structures, does not transfer easily to these other instances of the problem. This brings up to the following research question:

Research Question 4 *Can series of associative operators be made efficient no matter what the association pattern for all usage patterns?*

We present our solution to this question in Chapter 5. More precisely, our solution makes series of associative operations efficient regardless of the association pattern – and also provides efficient access to intermediate results. The key is to represent such a conceptual sequence of associative operations as an efficient sequence data structure. However, for some operators, such as the monadic bind, the type of the right argument depends on the type of the left argument. Efficient sequence data structures from the literature only support sequences where all elements have the same type, and hence they cannot be applied in a type-safe way in such situations. We introduce type aligned sequences which solve this problem. We demonstrate that our solution solves previously undocumented performance problems in Monadic FRP (Chapter 4), iteratees [Kiselyov, 2012], LogicT transformers [Kiselyov et al., 2005], free monads [Swierstra, 2008] and extensible effects [Kiselyov et al., 2013].

1.4 Origins of the chapters

- Chapter 2 was published earlier as:

  P. Klint and A. van der Ploeg (In alphabetical order). A Library for Declarative Resolution-independent 2d Graphics. In *Proceedings of the '13 International Symposium on Practical Aspects of Declarative Languages*, PADL '13, pages 1-18, 2013.

  It was co-authored by Paul Klint, his role was to help with the presentation and text, the ideas and implementation of the library where contributed by the author of this dissertation.

- Chapter 3 was published earlier as:

  A. van der Ploeg. Drawing Non-layered Trees in Linear Time. In *Journal of Software Practice & Experience (SP&E)*, Volume 44, Issue 12, pages 1467-1484, 2014.

- Chapter 4 was published earlier as:

  A. van der Ploeg. Monadic Functional Reactive Programming. In *Proceedings of the '13 Symposium on Haskell*, pages 117-128, 2013.

- Chapter 5 was published earlier as:

  A. van der Ploeg, O. Kiselyov. Reflection without remorse: revealing a hidden sequence to speed up monadic reflection. In *Proceedings of the '14 Symposium on Haskell*, pages 133–144, 2014.

  It was co-authored by Oleg Kiselyov, his role was to help with the presentation and text, and to research other occurrences of the problem. The solution, implementation and almost all text were contributed by the author dissertation.

All the above papers were peer-reviewed.

1.5 Other works by the author

P. Klint, B. Lisser and A. van der Ploeg. Towards a One-Stop-Shop for Analysis, Transformation and Visualization of Software (Invited Paper). In *Proceedings of the 4th international conference on Software Language Engineering*. SLE '13, pages 1-18. 2013.

B. Basten, J. van den Bos, M.A. Hills, P. Klint, A. Lankamp, B. Lisser, A. van der

Ploeg, T. van der Storm, J. Vinju. Modular Language Implementation in Rascal - Experience Report -. In *Science of Computer Programming, accepted for publication.*

M.A. Hills, A. Izamaylova, P. Klint, A. van der Ploeg, T. van der Storm, J.J. Vinju: The Rascal meta-programming language - a lab for software analysis, transformation, generation & visualization In: *Proceedings of ICT.Open 2011*, pages 353–358, 2011.

2

A Library for Declarative Resolution-Independent 2D Graphics[1]

Summary

The design of most 2D graphics frameworks has been guided by what the computer can draw efficiently, instead of by how graphics can best be expressed and composed. As a result, such frameworks restrict expressivity by providing a limited set of shape primitives, a limited set of textures and only affine transformations. For example, non-affine transformations can only be added by invasive modification or complex tricks rather than by simple composition. More general frameworks exist, but they make it harder to describe and analyze shapes. We present a new declarative approach to resolution-independent 2D graphics that generalizes and simplifies the functionality of traditional frameworks, while preserving their efficiency. As a real-world example, we show the implementation of a form of focus+context lenses that gives better image quality and better performance than the state-of-the-art solution at a fraction of the code. Our approach can serve as a versatile foundation for the creation of advanced graphics and higher level frameworks.

2.1 Introduction

The design of traditional 2D graphics frameworks, such as Java2D[2] and Processing[3], has been guided by what the computer can draw efficiently, instead of by how graphics can best be expressed and composed. This hinders the ease of programming 2D graphics, since it requires the programmer to express his ideas using the limited

[1]This chapter was published earlier as: P. Klint and A. van der Ploeg (In alphabetical order). A Library for Declarative Resolution-independent 2d Graphics. In *Proceedings of the '13 International Symposium on Practical Aspects of Declarative Languages*, PADL '13, pages 1-18, 2013.

[2]`http://docs.oracle.com/javase/6/docs/technotes/guides/2d/`

[3]`http://processing.org`

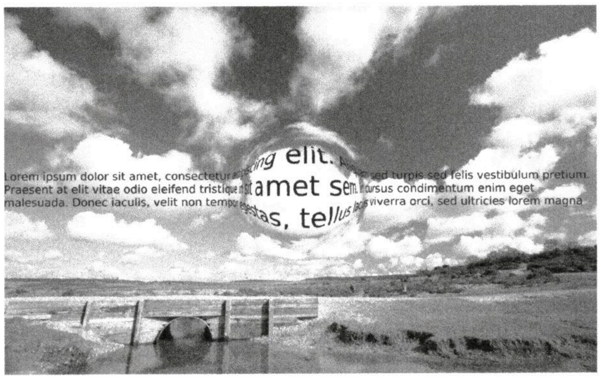


Figure 2.1: An example focus+context lens (zoomfactor = 2.5).

vocabulary that has emerged as a result of the focus on procedural optimization of such frameworks.

Suppose we have programmed a visualization in such a traditional framework and we now want to add a focus+context lens, such as the one shown in Figure 2.1. Since only affine transformations (that take parallel lines to parallel lines) are supported, we cannot add this transformation in a compositional way: it requires trickery or invasive modification.

Instead of worrying about such low-level details, it is desirable to program 2D graphics in a declarative way that is general, simple, expressive, composable and resolution-independent while still being efficient. Previous research on declarative graphics has yielded many elegant approaches to 2D graphics, but none of these exhibit all these traits. This not only restricts direct graphics programming, but it also hinders the creation of higher-level frameworks. For example, during our efforts on the Rascal figure library[Klint et al., 2011], a high-level framework for software visualization, we noticed that our design was influenced by the limitations of the procedural framework used and hence could not grow further in terms of expressiveness and compositionality.

We present a new declarative approach that generalizes and simplifies the functionality of traditional 2D graphics frameworks, while preserving their efficiency. This is achieved by a very effective mapping of our approach to an existing 2D graphics framework (which we will call the *graphics host*). Our approach allows more expressive freedom and can hence serve as a more versatile foundation for advanced 2D graphics and higher-level frameworks. It is available as a library called *Deform*[4] for Scala. Our contributions are:

- The motivation (Section 2.2) and design (Section 2.3) of a small, simple and powerful framework for resolution-independent 2D graphics that enables composability and expressiveness.

[4]`https://github.com/cwi-swat/deform`

- A way to implement and optimize this framework (Section 2.4) by mapping it to a readily-available, highly optimized graphics host. This includes optimizations to speed up this mapping and a way to support clipping so that large scenes can be rendered in real-time.
- An implementation of focus+context lenses that is faster and gives better image quality than the state-of-the-art approach (Section 2.5). This also acts as a validation of our work.

We discuss open questions in Section 2.6 and conclude in Section 2.7.

2.2 Exploring the Design Space

We now discuss design choices for declarative 2D graphics frameworks and to guide our choices, we use the following design goals:

- *Simplicity*: The programmer should not be overwhelmed by concepts and functions described in inch-thick manuals.
- *Expressivity*: Arbitrary graphics can be expressed in a *natural* way, without the need to encode them in lower-level concepts.
- *Composability*: Graphics can be composed and transformed in general ways.
- *Resolution-independence*: Graphics can be expressed independent of resolution, so that they can be rendered at any level of detail.
- *Analyzability*: The concrete geometry of a shape can be obtained, for example as a list of lines and Bézier curves, so that we can define functions that act on this information to create derived graphics.
- *Optimizability*: Efficient algorithms for 2D graphics can be re-used.

Our analysis now focuses on how to represent *shapes*, *textures* and *transformations*, in the way that has the best fit with our design goals.

2.2.1 Shapes

Most frameworks offer a fixed set of geometric constructs, such as lines, Bézier curves and circle segments, that can be used to describe the *border* of shapes. For example, a regular polygon with k vertices can be expressed as follows:

$$\begin{aligned} \mathit{regpolyg}(k) &= [\mathit{line}(\mathit{onCircle}(i \times p), \mathit{onCircle}((i+1) \times p)) \mid i \leftarrow [0 \ldots k-1]] \\ &\quad \textbf{where } \mathit{onCircle}(x) = \langle \sin(x), \cos(x) \rangle, \;\; p = (1/k) \times 2 \times \pi \end{aligned}$$

Here $\langle x, y \rangle$ denotes a point in $\mathbb{R}^2$. A downside of this approach is that shapes that are not compositions of such geometric constructs, such as sine waves, cannot be expressed. Instead, they have to be approximated *when specifying the shape*, which does not give a resolution-independent description of the shape.

A second approach is to describe the border of a shape as a *parametric* curve: a function from $\mathbb{R}$ to $\mathbb{R}^2$. For example, the border of the unit circle can be described by $c(t) = \langle \sin(t \times 2 \times \pi), \cos(t \times 2 \times \pi) \rangle$ on the interval $[0, 1]$. This can be seen as a generalization of using a fixed set of geometric constructs: each geometric construct can be described by a parametric curve and hence a combination of geometric constructs gives rise to a piecewise defined function. For this reason the expression of a regular polygon with k vertices is exactly the same as when using a fixed set of geometric constructs. Although a parametric description does not immediately give an analyzable description of the shape, we can sample the (resolution-independent) function to obtain such a description.

The third and final approach is to describe a shape *implicitly*: as a function that given a point in $\mathbb{R}^2$ tells us whether the point is inside the shape or not. For example, the implicit representation of the unit circle is $c(p) = |p| \leqslant 1$, where $|p|$ denotes the Euclidian norm. A downside of this approach is that it is often hard to encode a shape in this way. For example, as noted in [Karczmarczuk, 2002], it requires an arcane insight to understand that the following also represents a regular polygon with k vertices.

$$\begin{aligned}
&\mathit{regpolyg} \quad (k, \langle x, y \rangle) = (x - j) \times (\sin(q + p) - i) - (\cos(q + p) - j) \times (y - i) \leqslant 0 \\
&\qquad\qquad\quad \textbf{where}\, p = 2 \times \pi / k, \quad q = p \times \lfloor \mathit{atan2}(y, x) / p \rfloor, \quad i = \sin(q), \quad j = \cos(q)
\end{aligned}$$

It is also hard to analyze a shape that is described in this way, since we do not have a representation of the border of the shape.

If we could automatically switch between the parametric and implicit representations we would not have to make a choice between them. However, transforming a parametric representation into an implicit one or vice-versa is non-trivial, especially when the functions are not limited to a certain class. In fact, these are well-known and thoroughly studied problems [Hoffmann, 1993]. In general, exact conversion is possible for certain classes of functions [Sederberg et al., 1984], while other classes of functions require approximate techniques [Dokken and Thomassen, 2003]. Since the implicit representation makes it hard to express and analyze shapes, and since it is hard and computationally expensive to automate the conversion between the two representations we have chosen to describe shapes parametrically.

2.2.2 Textures

Most frameworks offer a fixed set of textures, such as fill colors, images and gradients. Another approach is allow arbitrary textures by specifying the colors of its pixels, but this is not a resolution independent approach. A general, resolution independent way to describe a texture, and the one that we adopt, is by a function that given a point returns the color of the texture at that point [Elliott, 2001; Karczmarczuk, 2002]. Notice that this way of expressing textures bears resemblance to implicitly defined shapes: implicitly defined shapes are functions of type $\mathbb{R}^2 \rightarrow \mathit{Boolean}$, whereas such textures are functions of type $\mathbb{R}^2 \rightarrow \mathit{Color}$.

		Traditional	Func. image synthesis	Vertigo	Deform
Shapes	Fixed	●			
	Parametric			●	●
	Implicit		●		
Textures	Fixed/pixels	●			
	Function		●		●
Transforms	Affine	●			
	Function			●	●
	Function^{-1}		●		●

Table 2.1: Design choices for graphics libraries.

2.2.3 Transformations

Typically, graphics frameworks offer only affine transformations, such as translation, rotation and scaling. Although these transformations cover many use cases, they preclude a whole range of interesting transformations, such as focus+context lenses. A more expressive model is to describe transformations simply as a function from $\mathbb{R}^2$ to $\mathbb{R}^2$.

Parametrically described shapes then require the *forward* transformation, while textures and implicitly defined shapes require the *inverse* transformation. For example, to translate a parametrically defined shape to the right, we define a function that given a parameter first gets the corresponding point on the border of the shape and then applies the forward transformation to that point, which moves the point to the right. To translate a texture to the right, we define a function that given a point first applies the inverse transformation, which moves the point to the left, and then queries the texture at that point. In the same fashion, the inverse transformation is also needed to transform implicitly defined shapes.

If we limit ourselves to affine transformations, obtaining both directions of a transformation is not a problem since such transformations are easily inverted. However, if we allow arbitrary transformations we need to either describe all shapes implicitly and use only the inverse transformation, making it harder to describe shapes, or describe shapes parametrically in which case we need *both* the forward transformation and the inverse transformation, making it harder to describe transformations. We conjecture that shapes are more likely to be application-specific than transformations, which can often be reused. Hence, we have chosen to represent shapes parametrically and require a definition of both directions for transformations.

2.2.4 Comparison

As a comparison, Table 2.1 lists the choices made by us and other frameworks. *Traditional* frameworks, like as Java2D, Processing and many others, limit the expressivity

Constructor	*Type*
path	$(\mathbb{R} \to \mathbb{R}^2) \to Path$
shape	$[Path] \to Shape$
analyze	$Path \times (ConcreteGeom \to A) \to A$
	where $A \in \{Path, Shape, Texture, TexturedShape, Transform\}$
color	$\mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to Color$
texture	$(\mathbb{R}^2 \to Color) \to Texture$
fill	$Shape \times Texture \to TexturedShape$
transformation	$(\mathbb{R}^2 \to \mathbb{R}^2) \times (\mathbb{R}^2 \to \mathbb{R}^2) \to Transformation$

Table 2.2: Constructors and functions. $[A]$ indicates a list of As.

of the programmer by only providing support for the most common use cases. Many declarative graphics frameworks[5] make the same choices [Finne and Jones, 1995; Matlage and Gill, 2009]. *Functional image synthesis* frameworks, such as Pan [Elliott, 2001] and Clastic [Karczmarczuk, 2002], are based on the notion that an image is simply a function from a point to a color. This allows the elegant definition of many interesting visual mathematical graphics but precludes real-life graphics, since the requirement of implicitly defined shapes makes hard to define complex shapes such as letters. *Vertigo* [Elliott, 2004] is an elegant declarative framework for the geometric modeling of 3D shapes, without texturing. In *Deform* we have chosen a combination of design decisions that has not yet been explored: parametric shapes, textures as functions and general transformations. In the rest of this chapter we show that this allows us to define a simple, general and resolution-independent framework which is applicable to real-life graphics.

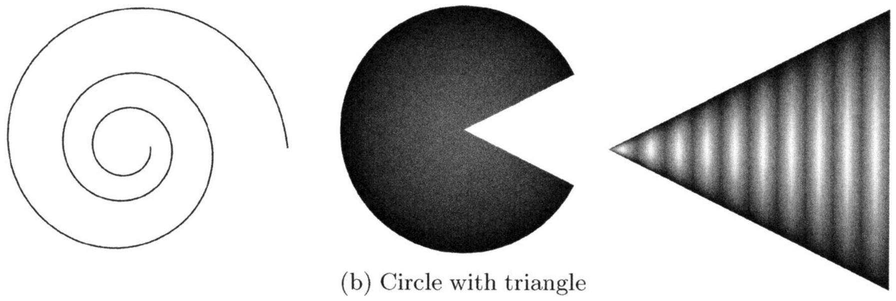
2.3 Design

It is time to present our approach and illustrate its usage via examples. The basic unit of our framework is a *TexturedShape*, that describes a shape and the texture of its interior. An expression constituting a list of such textured shapes is first *created* using the constructors given in Table 2.2 and then *displayed* by a render function which interprets the constructors and produces an image. We will now show how to express shapes, textures and transformations in this way. Our examples were programmed in Scala and then hand-transformed into a custom notation which should be easy to understand. The examples use the constructors in Table 2.2 and some library functions of Deform, both of which will be explained when used.

2.3.1 Shapes

The basis for describing shapes is the *path* constructor, which takes a parametric description of the border of the shape, a function of type $\mathbb{R} \to \mathbb{R}^2$. To allow omission

[5]Unfortunately, space limitations do not allow a more extensive discussion.



(a) A simple spiral

(b) Circle with triangle subtracted

(c) A filled triangle

Figure 2.2: Basic examples

of the domain of this function, it simply must be $[0,1]$. The *shape* constructor can then be used to create a shape from a list of *closed* paths, paths of which the start and end points are the same. If one of the paths is not closed, then it does not define an area and a run-time error will be thrown. A point is then inside the shape if it is inside any of its closed paths.

As a basic example, consider a circle:

$$\mathit{circ} = \mathit{shape}([\mathit{path}(\lambda t \rightarrow \langle \sin(t \times 2 \times \pi), \cos(t \times 2 \times \pi)\rangle)])$$

The coordinate system of our framework is as follows: if the screen is square then the north west corner of the screen is $\langle -1,-1\rangle$ and the south east corner is $\langle 1,1\rangle$. If the screen is non-square the range of the longest axis is adopted so that graphics maintain their aspect-ratio. An example of a more complex path is the spiral shown in Figure 2.2a:

$$\begin{aligned}\mathit{spiral} = \mathit{path}(\lambda t \quad &\rightarrow \langle f \times \cos(s), f \times \sin(s)\rangle \\ &\textbf{where } f = 1/50 \times e^{s/10}, \quad s = 6 \times \pi \times (1+t)\end{aligned}$$

Paths themselves cannot be drawn as they do not define an area. Hence, to produce a drawing of this spiral we use the *stroke* library function to convert this path to a shape given the width of the "pen":

$$\mathit{stroke}(\mathit{spiral}, 1/200)$$

We do not have to explicitly define a parametric representation for each shape. Instead, we provide library functions that mimic the geometric constructs found in traditional libraries. For example, we can create a triangle as follows:

$$\begin{aligned}\mathit{triangle} \quad &= \mathit{shape}([\mathit{join}([\mathit{line}(a,b), \mathit{line}(b,c), \mathit{line}(c,a)])]) \\ &\textbf{where } a = \langle 0,0\rangle, \quad b = \langle 1, \tfrac{1}{2}\rangle, \quad c = \langle 1, -\tfrac{1}{2}\rangle\end{aligned}$$

To define functions which act on the geometry of a path, such as the *stroke* function, we offer the *analyze* constructor which takes a path and a function transforming the concrete geometry of the path, namely a list of lines and Bézier curves, into a path, shape, texture, textured shape or transformation. To ensure resolution-independence, *analyze* is a constructor rather than a function: in this way we delay the sampling of the path until we know the desired resolution, namely when the renderer runs. We also use this constructor to define resolution independent *constructive solid geometry operations* on shapes, set operations such as union and intersection operating on the set of points inside a shape. The implementation of these operations involves analyzing the intersections between the concrete geometry of both shapes. As an example, the shape in Figure 2.2b can be obtained as follows:

$$\mathit{pacman} = \mathit{subtract}(\mathit{circ}, \mathit{triangle})$$

2.3.2 Textures

To declare the interior of a shape, a texture can be created with the *texture* constructor, which requires a function from a point to a color. A *color* is a value with four numbers, all in the range $[0, 1]$, namely red, green, blue and alpha (transparency). For example, consider the following colors:

$$\mathit{red} = \mathit{color}(1,0,0,1), \mathit{black} = \mathit{color}(0,0,0,1), \mathit{yellow} = \mathit{color}(1,1,0,1)$$

We can now create a radial gradient as follows:

$$\mathit{radgrad} = \mathit{texture}(\lambda\langle x, y\rangle \rightarrow \mathit{lerp}(\mathit{red}, x^2 + y^2, \mathit{black}))$$

Where *lerp* performs linear interpolation of two colors on each of the four numbers. A *TexturedShape* can then be created using the *fill* constructor. For example, Figure 2.2b shows:

$$\mathit{fill}(\mathit{pacman}, \mathit{radgrad})$$

As another example of defining textures in our framework, consider the interior of the triangle shown in Figure 2.2c. For this texture, we first declare a one-dimensional cyclic gradient that cycles between red and yellow:

$$\begin{aligned} \mathit{gradient}(x) \; &= \textbf{if } l \leqslant \tfrac{1}{2} \textbf{ then } \mathit{lerp}(\mathit{red}, 2\times l, \mathit{yellow}) \\ &\quad \textbf{else } \mathit{lerp}(\mathit{yellow}, 2\times(l - \tfrac{1}{2}), \mathit{red}) \\ &\quad \textbf{where } l = x - \lfloor x \rfloor \end{aligned}$$

We can then define the filling of the triangle as follows:

$$\mathit{tritex} = \mathit{texture}(\lambda\langle x, y\rangle \rightarrow \mathit{lerp}(\mathit{gradient}(x\times 10), (2\times|y|/x)^2, \mathit{black})$$

Where $x\times 10$ repeats the gradient ten times on the horizontal $[0, 1]$ interval and the linear interpolation argument[6] $(2\times|y|/x)^2$ ensures that the color becomes darker closer to the vertical border of the triangle. A further survey of the power of this way of describing textures is beyond the scope of this chapter, for some fascinating examples see [Elliott, 2001] and [Karczmarczuk, 2002].

[6]When $x = 0$, $|y|/x$ will be ∞ or not-a-number, which will cause *lerp* to return black.

2.3.3 Transformations

The *transformation* constructor can be used to describe arbitrary transformations and requires the forward transformation function and its inverse. For example, we can define a scaling transformation as follows:

$$\begin{aligned} \mathit{scale}(s_x, s_y) = \mathit{transformation}(&\lambda\langle x, y\rangle \rightarrow \langle s_x \times x, s_y \times y\rangle, \\ &\lambda\langle x, y\rangle \rightarrow \langle x/s_x, y/s_y\rangle) \end{aligned}$$

We can use this transformation to scale our previous examples. For example, to make our filled triangle half as big, we can do the following:

$$\mathit{transform}(\mathit{scale}(1/2, 1/2), \mathit{fill}(\mathit{triangle}, \mathit{tritex}))$$

Where the *transform* function is expressed as follows:

$$\begin{aligned} &\mathit{transform}(\mathit{transformation}(f, f^{-1}), \mathit{path}(p)) = \mathit{path}(f \circ p) \\ &\mathit{transform}(f, \mathit{shape}(l)) = \mathit{shape}([\mathit{transform}(f, p) \mid p \leftarrow l]) \\ &\mathit{transform}(\mathit{transformation}(f, f^{-1}), \mathit{texture}(t)) = \mathit{texture}(t \circ f^{-1}) \\ &\mathit{transform}(f, \mathit{fill}(s, t)) = \mathit{fill}(\mathit{transform}(f, s), \mathit{transform}(f, t)) \end{aligned}$$

The only constraint on a transformation is that it must be continuous, otherwise it would be possible to transform a closed path (defining an area) into an open path (not defining an area). As an example of a non-affine transformation consider the "wave" transformation shown in Figure 2.3a:

$$\mathit{wave} = \mathit{transform}(\lambda\langle x, y\rangle \rightarrow \langle x + \sin(y), y\rangle), \lambda\langle x, y\rangle \rightarrow \langle x - \sin(y), y\rangle)$$

These transformations can be composed using the following *compose* function, which uses the well-known rule $(f \circ g)^{-1} = g^{-1} \circ f^{-1}$.

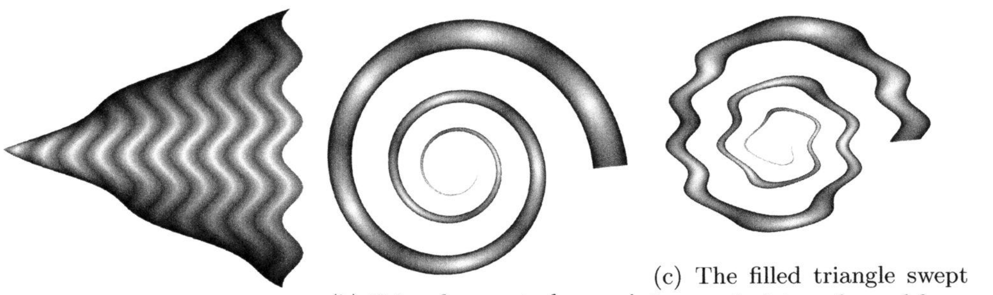
$$\mathit{compose}(\mathit{transform}(f, f^{-1}), \mathit{transform}(g, g^{-1})) = \mathit{transform}(f \circ g, g^{-1} \circ f^{-1})$$

A benefit of having both directions of a transformation is that we can also transform *transformations*. For example, if we have a rotation transformation and we want to change the center of rotation, we can achieve this by transforming the rotation by a translation. This is done by first applying the inverse translation, then the rotation and then the forward translation. In general, we can transform any transformation by another transformation as follows:

$$\begin{aligned} &\mathit{transform}(t, r) = \mathit{compose}(t, \mathit{compose}(r, \mathit{inverse}(t))) \\ &\qquad\qquad \textbf{where } \mathit{inverse}(\mathit{transform}(f, f^{-1})) = \mathit{transform}(f^{-1}, f) \end{aligned}$$

As an example, we can transform our wave transformation to produce smaller waves:

$$\mathit{scaledWave} = \mathit{transform}(\mathit{scale}(1/30, 1/30), \mathit{wave})$$



(a) Wave transformed triangle.

(b) Triangle swept along spiral.

(c) The filled triangle swept by a spiral transformed by a wave.

Figure 2.3: Non-affine transformation examples

Applying this transformation to our filled triangle produces Figure 2.3a.

Another example of a non-affine transformation is a "sweep": mapping the [0,1] interval on the x-axis to a given path. For example, by first scaling our filled triangle to make it thinner we can obtain Figure 2.3b as follows:

$$\mathit{fspir} = \mathit{transform}(\mathit{compose}(\mathit{sweep}(\mathit{spiral}), \mathit{scale}(1, 1/40)), \mathit{ftriangle})$$

Other papers [Karczmarczuk, 1999; Elliott, 2004] have shown how to implement the sweep transformation when only the forward transformation is required, we now show how to handle both directions of this transformation. To define this transformation in a resolution-independent way, we define it as a function which takes the concrete geometry of the path and returns a transformation. Using the *analyze* constructor, we make this function into a transformation.

To prevent changes in speed along the path, we want the norm of the derivative to be constant along the path. To this end, we reparameterize the concrete geometry of the path to a new geometrical description, q, with the same shape and a constant norm of the derivative, using an algorithm such as [Casciola and Morigi, 1996]. The forward transformation can then be expressed as follows:

$$\lambda\langle x, y\rangle \rightarrow q(x) + y \times \widehat{q'(x)}$$

Here $\hat{x}$ denotes a normalized vector and q' is the derivative of q.

The inverse transformation works by finding the closest point on the path to the point that is to be transformed. The horizontal coordinate is then the parameter at that point on the path, and the vertical coordinate is the distance of the point to be transformed from the path. More precisely:

$$\lambda v \rightarrow \langle t, \mathit{sgn}(q'(t)) \times |q(t) - v|\rangle \ \textbf{where}\ t = f(v)$$

Here *sgn* is the sign function and f computes the parameter of the closest point on q to a given point, using an algorithm such as [Ma and Hewitt, 2003]. As a final example of the compositionality this framework gives us, we transform the swept triangle using our wave transformation to obtain Figure 2.3c:

$$transform(scaledWave, fspir)$$

2.4 Implementation and Optimization

Our approach can be efficiently implemented by mapping it to a graphics host. We first describe a basic implementation and then introduce some extensions to allow more optimizations. Finally, we show how we can support clipping and discuss potential further optimization. The implementation of Deform as sketched in this section is surprisingly concise and simple and consists of just 983 lines of Scala code.

2.4.1 Basic implementation

The main function to implement is the *render* function, which acts as an interpreter for the constructors that may occur in a *TexturedShape*. The pipeline of the *render* function is shown in Figure 2.4 and is organized as follows; A *TexturedShape* is produced by the user program and its shape is then translated into geometry, i.e., lines and Bézier curves, which are in turn translated to their equivalent representations in the graphics host. The graphics host then fills the shape, producing a raster telling us which pixels are inside the shape. We then simply iterate over these pixels and call the corresponding texture function for each pixel, producing a color raster which is then sent to the display.

The *toBézier* function in this pipeline is also used to interpret *analyze* constructors, namely to generate the concrete geometry which is fed to the function argument of the constructor. We currently use a simple implementation of this function: we sample the function until the samples are so close to each other that the error is smaller that the size of a pixel. Afterwards, the samples are joined by lines.

2.4.2 Special cases

We optimize the basic implementation by intercepting special cases and mapping them to the corresponding functionality of the graphics host. We add a new constructor for each special case, which are shown in Table 2.3. Several of these new constructors were presented earlier as functions and by transforming them into constructors the *render* function can recognize them and act accordingly. We now discuss the special cases for shapes, textures and transformations.

Shapes

The first special case for shapes concerns paths that consist of lines and Bézier curves. It is of course wasteful to use a combination of lines and Bézier curves, only to later approximate it with other lines and Bézier curves. Hence, we extend our *Path* type

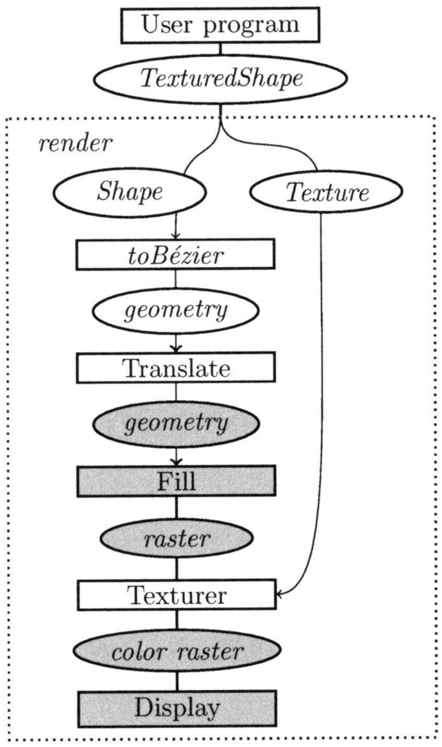


Figure 2.4: Rendering pipeline. Gray indicates functionality from the graphics host.

with extra constructors for these types of paths and a constructor for *join*, so that the *toBézier* function can immediately use these descriptions without sampling.

The second special case for shapes deals with constructive solid geometry operations. The default implementation of these operations is to obtain a concrete geometry of the shapes using *toBézier* and then analyze intersections to produce the new shape. In the case of union or symmetric difference we can skip this analysis. The union of a set of shapes can be implemented by supplying the set of shapes to the fill function of the graphics host and using the *non-zero fill rule.* This tells the renderer to fill any pixel that is inside at least one of the shapes, effectively rendering the union of the shapes. Analogously, we can render the symmetric difference of a list of shapes by using the *even-odd fill rule*, which states that a pixel should be filled if it is inside an odd number of shapes.

Textures

If the graphics host has support for a texture, we would like to make use of these optimized capabilities, because then we can completely skip the Texturer step in the pipeline. Hence, we include the constructor *nativeTexture* for these cases, which takes a function that given an affine transformation gives the specific representation for the graphics host of the transformed texture and a regular texture function for use when the transformation of the texture is not affine.

Transformations

If a transformation is affine and the path consists of lines and Bézier curves, we transform the geometry directly, instead of by sampling a function. The constructor *affineTransformation* represents such an affine transformation by two matrices (the specification of this type is left open), one for the forward transformation and one for the inverse transformation. We also change the *transform* function into a constructor so that the *toBezier* function can intercept this special case. The *compose* function is also adapted to intercept the special case of composing an affine transformation with another affine transformation, which can be done using matrix multiplication instead of function composition, saving computations when points are transformed.

Performance

Note that in traditional frameworks such as Java2D or Processing, the special cases presented above are the *only* things that are expressible. Thus, the interception of these special cases guarantees that drawings that could also be produced using such a library are approximately as fast. We verified this by generating equivalent Java2D and Deform code in which 100,000 shapes (letters) were rendered, each with their own native texture and affine transformation. The Deform code performed 0.8% slower than the direct Java2D calls. This minor difference in speed is due to the fact that the Deform code first builds an intermediate representation of the textured shapes.

Constructor	*Type*
line	$\mathbb{R}^2 \times \mathbb{R}^2 \to Path$
quadBezier	$\mathbb{R}^2 \times \mathbb{R}^2 \times \mathbb{R}^2 \to Path$
cubicBezier	$\mathbb{R}^2 \times \mathbb{R}^2 \times \mathbb{R}^2 \times \mathbb{R}^2 \to Path$
join	$[Path] \to Path$
union	$[Shape] \to Shape$
symdiff	$[Shape] \to Shape$
nativeTexture	$(Matrix \to NativeTextureDesc) \times (\mathbb{R}^2 \to Color)$ $\to Texture$
transformation	$(\mathbb{R}^2 \to \mathbb{R}^2) \times (\mathbb{R}^2 \to \mathbb{R}^2) \to Transformation$
transform	$Transformation \times A \to A$ **where** $A \in \{Path, Shape, TexturedShape, Transform\}$
affineTransformation	$Matrix \times Matrix \to Transformation$
pathbb	$(\mathbb{R} \to \mathbb{R}^2) \times BBox \to Path$
transformationbb	$(\mathbb{R}^2 \to \mathbb{R}^2) \times (\mathbb{R}^2 \to \mathbb{R}^2) \times (BBox \to BBox)$ $\to Transformation$

Table 2.3: Additional constructors for special cases.

2.4.3 Clipping

For large scenes, involving many shapes, a valuable optimization is *clipping*: determining the bounding boxes of shapes and then ignoring the shapes that are not in view. However, since in our framework shapes and transformations can be arbitrary functions, it is impossible to discover the bounding box of a shape without sampling it.

For this reason we add two new constructors: one to declare a path and its bounding box (the specification of this type is left open) and one to declare a transformation and also a function to forwardly transform a bounding box. In this way the user can optionally give the bounding boxes of transformed shapes. If the bounding boxes are not supplied, the shapes will simply not profit from clipping. In Deform, all library functions to construct paths and transformations also deal with bounding boxes. For example, lines and Bézier curves get the bounding box induced by their (control) points and *join* produces the smallest bounding box that contains the bounding boxes of its arguments. Affine transformations transform a bounding box by transforming each of its vertices. We currently use axis-aligned bounding boxes, but it is also possible to use non-axis-aligned bounding boxes that fit the shapes more tightly, at the cost of more computations.

2.4.4 Potential optimization

A potential optimization might be to speed the *toBézier* function by using techniques from the field of curve fitting. We could do the sampling and fitting in parallel, by modifying a curve fitting algorithm such as [Schneider, 1990]. We can then stop the sampling earlier if the samples we take lie close enough to the current approximation. We can also use the parameter of each point to improve the speed of our approximation since this is often useful information for curve fitting algorithms [Schneider, 1990]. Finally, curve fitting algorithms often estimate a derivative of the shape, so if we numerically compute the derivative, or supply it using an automated differentiation system [Elliott, 2009a], we can also use this information to more quickly find an approximation of the curve.

2.5 Case study: Focus+context Lenses

As a real world example of how this framework enables advanced, resolution-independent computer graphics techniques in a compositional way, we show how to implement the form of focus+context lenses that are presented in [Carpendale and Montagnese, 2001], which have been shown to be useful in human computer interaction [Pietriga et al., 2010]. A focus+context lens, such as the one in Figure 2.1, is a transformation that magnifies a part of the space (the focus area) and shows how this magnified part fits into the rest of the space (the context) through a deformation. We compare our implementation to the previous implementation of this form of focus+context lenses [Pietriga et al., 2010]. Our implementation is slightly harder, since we require both directions of the transformation. As we will show, this effort is well spent since it yields a faster implementation that gives better image quality at a fraction of the code.

2.5.1 Implementation

We first consider the *inverse* transformation as presented in [Carpendale and Montagnese, 2001; Pietriga et al., 2010]. Figure 2.5 shows the elements of a lens: r_f is the radius of the focus area, r_l is the radius of the lens and we define m as the magnification factor. The inverse transformation is then defined as follows:

$$l^{-1}(v) = \begin{cases} v/m & |v| < r_f \\ \frac{v}{|v|} \times n^{-1}(|v|) & r_f < |v| < r_l \\ v & \text{otherwise} \end{cases}$$

Where n^{-1} is the function that describes the deformation, by giving the new norm, i.e., distance from the center of the lens, for the point to be transformed and is a continuous, monotonically increasing function from $[r_f, r_l]$ to $[r_f/m, r_l]$:

$$n^{-1}(d) = \frac{d}{(1 - p(z)) \times (m - 1) + 1} \quad \textbf{where } z = (d - r_f)/(r_l - r_f)$$

Here z describes how far into the deformation area the point is, with zero if the point is on the border of the magnification area and one if it is on the border of the context area. The *profile* function, p, describes the shape of the deformation and can be

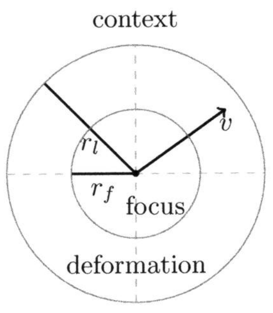


Figure 2.5: Lens elements.

chosen freely as long as it is a continuous, monotonically increasing function from $[0,1]$ to $[0,1]$, such as the identity function. Another variation point is which norm to use to compute $|v|$, which decides the shape of the lens. In general it is possible to use any L^P norm, which are of the form $\sqrt[p]{x^p + y^p}$. The lens is circular with L^2 and with L^∞ the norm resolves to $\max(x, y)$ and the lens is square. The example in Figure 2.1 uses the Euclidian norm and a Gaussian profile function and Figure 2.6 shows two more Deform screenshots of other lenses in action.

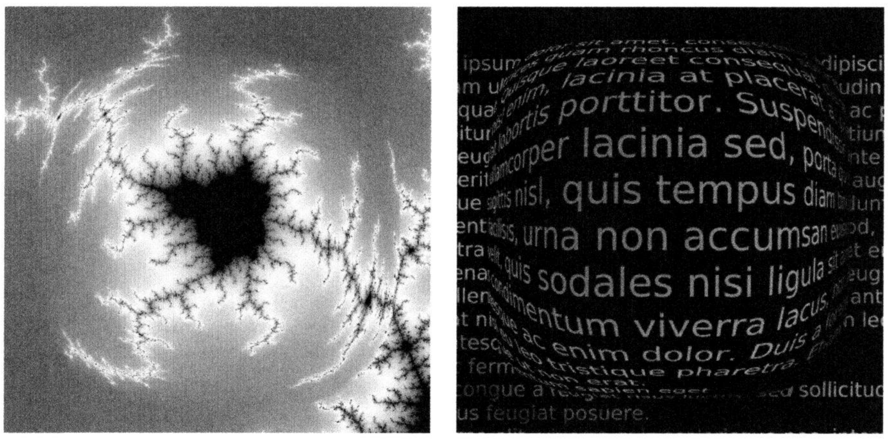
We now need to derive the *forward* transformation from this inverse transformation. If we have the inverse of the function n^{-1}, then the forward transformation can be expressed as follows:

$$l(v) = \begin{cases} v \times m & |v| < r_i/m \\ \frac{v}{|v|} \times n(|v|) & r_f/m < |v| < r_l \\ v & \text{otherwise} \end{cases}$$

However, for many profile functions, there is no analytic solution for the inverse of n^{-1}. Luckily, n^{-1} is a continuous monotonically increasing function, so we can implement $n(t)$ by numerically searching for the x such that $n^{-1}(x) = t$. We use Newton's method for this, since it is very efficient at finding the roots of monotonic functions. This method requires the derivative of n^{-1}, which can be constructed using the derivative of the profile. In this way only the profile function and its derivative are needed when creating a lens with a different profile.

2.5.2 Comparison

The previous implementation [Pietriga et al., 2010] of this form of focus+context lenses is in the Zoomable Visual Transformation Machine (ZVTM) [Pietriga, 2005] framework for zoomable user interfaces. The advantage of their approach to implementing these lenses is that it is very loosely coupled with the graphics host, and is thus applicable in many graphical frameworks. In our approach these lenses can be added easily and this yields a better implementation in terms of length of code, speed and image quality.


lacinia at placerat
porttitor. Suspendisse
ullamcorper lacinia sed, porta
sagittis nisl, quis tempus diam
iaculis, urna non accumsan
quis sodales nisi ligula sit
condimentum viverra lacus
ac enim dolor. Duis
tristique pharetra
sollicitu
us feugiat posuere.

(a) L^3 norm, linear profile (b) L^4 norm, quadratic profile

Figure 2.6: Different types of lenses in action

Code size

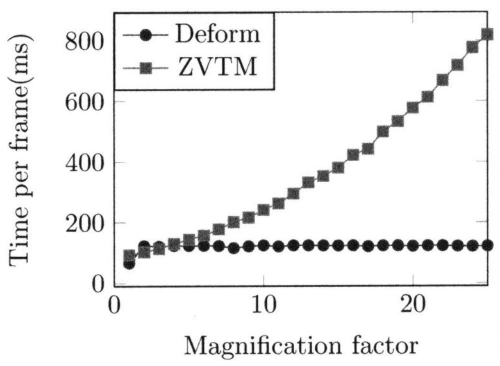
In the ZVTM implementation, defining the lenses requires about 700 lines of code, and each new lens (with a different norm or profile) requires about 100 lines of code [Pietriga et al., 2010]. In our declarative framework, the implementation of these lenses requires 43 lines of code, including the definition of the (reusable) numeric approximation code, while defining a new lens can be done in a single line of code. For example, a rounded square lens with a quadratic profile (with derivative $2\times x$), as shown in Figure 2.6b, is declared as follows:

$$lens(\lambda\langle x, y\rangle \to \sqrt[4]{x^4 + y^4}, \lambda x \to x^2, \lambda x \to 2\times x)$$

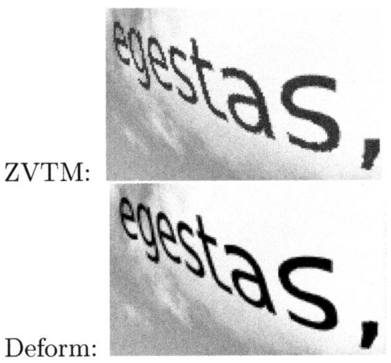
Performance

As a performance comparison, we implemented the setup shown in Figure 2.1 in both Deform and ZVTM and measured the time it took to render a single image at different magnification factors. This was chosen because it is a simple example of a combination of shapes (text) and a texture (bitmap image). The entire picture was 1600x1000 pixels big and the lens had a focus radius of 100 pixels and a lens radius of 200 pixels. Note that both ZVTM and Deform run on the JVM and are built on top of Java2D. Figure 2.7a shows the results of our measurements on an Intel i7 2.8GHz CPU running OpenJDK 1.11.3. All measurements are the average of 100 runs.

We can see that in ZVTM the magnification factor has a huge impact on performance, whereas in Deform it has no effect at all. This is because ZVTM does not feature non-affine transformations in general and uses a trick to achieve focus+context lenses; It renders the lens area *twice*: once without magnification and once with magnification. Afterwards, both renderings are sampled to produce the lens area. The second, magnified rendering uses a buffer of width and height $2\times m\times r_l$. Hence the



(a) Difference in speed



(b) Difference in rendering quality

Figure 2.7: Performance and image quality comparison.

amount of pixels in this buffer is $(2 \times m \times r_l)^2$, which explains the quadratic growth of the ZVTM rendering time.

Image quality

As a final comparison, we consider the image quality of both approaches as shown in Figure 2.7(b). This notable difference in image quality is caused by the fact that Deform performs the discretization of shapes and textures later. ZVTM performs the discretization *before* applying the lens, while Deform performs the discretization *after* applying the lens. Hence Deform does not suffer from aliasing artifacts.

2.6 Discussion

While our framework is very expressive, it currently does not support post-processing image filters such as blurs. These filters are computationally very expensive and require low-level optimizations for real-time performance. Halide [Ragan-Kelley et al., 2012] is an example of a language that is specifically designed for such filters; the programmer gives a concise declarative description of the filter along with a schedule that states how the filter must be implemented. This yields very good results, outperforming hand tuned assembly code in some cases. It would be interesting to explore how the Halide way of describing filters can be fitted into our framework.

Another open question is how we can exploit the massive power that is available via GPUs: which paths, transformations and textures can be executed on the GPU and how? How can these parts work together with functionality that cannot be executed on the GPU? Answering these questions will lead to a truly high-performance implementation of Deform.

2.7 Conclusion

We have presented a novel declarative framework for resolution-independent 2D graphics that is simple, expressive and composable while still being applicable to real-life graphics. We have shown how to implement this framework such that it easily maps to readily available, highly-optimized procedural graphics libraries and have also shown how this framework can support clipping, so that it is possible to render very large scenes. We have shown a simple benchmark that shows that our framework is as fast as directly using the graphics host, thanks to the interception of special cases. As a real-world example, we have implemented focus+context lenses. The result is faster and smaller than the state-of-the-art implementation and has better image quality. Our framework liberates the programmer from the limitations of traditional frameworks and we expect that it forms an excellent foundation for creating resolution-independent graphics and higher-level visualization tools in a wide range of domains.

Acknowledgements

We thank Robert van Liere and Tijs van der Storm for their helpful comments.

3

Drawing Non-layered Tidy Trees in Linear Time[1]

Summary

The well-known Reingold-Tilford algorithm produces tidy *layered* drawings of trees: drawings where all nodes at the same depth are vertically aligned. However, when nodes have varying heights, layered drawing may use more vertical space than necessary. A *non-layered* drawing of a tree places children at a fixed distance from the parent, thereby giving a more vertically compact drawing. Moreover, non-layered drawings can also be used to draw trees where the vertical position of each node is given, by adding dummy nodes. In this chapter we present the first linear time algorithm for producing non-layered drawings. Our algorithm is a modification of the Reingold-Tilford algorithm, but the original complexity proof of the Reingold-Tilford algorithm uses an invariant that does not hold for the non-layered case. We give an alternative proof of the algorithm and its extension to non-layered drawings. To improve drawings of trees of unbounded degree, extensions to the Reingold-Tilford algorithm have been proposed. These extensions also work in the non-layered case, but we show that they then cause a $O(n^2)$ run-time. We then propose a modification to these extensions that restores the $O(n)$ run-time.

3.1 Introduction

In many fields, trees are a much used abstraction. The understanding of trees is greatly improved by visualizing them and hence many types of tree drawings have been proposed [Tollis et al., 1998; Johnson and Shneiderman, 1991; Kleiberg et al., 2001]. In this chapter, we focus on classical node-link diagrams, an example of which

[1]This chapter is was published earlier as: A. van der Ploeg. Drawing Non-layered Trees in Linear Time. In *Journal of Software Practice & Experience (SP&E)*, Volume 44, Issue 12, pages 1467-1484, 2014.

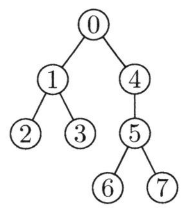


Figure 3.1: Example node-link diagram of a tree.

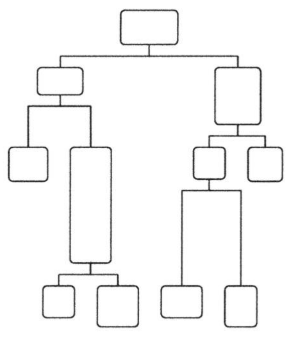
is shown in Figure 3.1. Usually, we are mainly interested in showing the structure of the tree and hence all the nodes can have the same width and height as shown in Figure 3.1. Sometimes, we also want to show some information inside each node or in the dimensions of each node. Examples of this are Tableau style proof trees, parse trees of (formal) languages, class diagrams in software engineering and Polymetric views [Lanza and Ducasse, 2003b]. The latter are inheritance diagrams of software systems where the width and height of each node signifies a software metric of the corresponding class, such as the lines of code or number of methods. In these situations, the width and height of each node may vary.

Trees with nodes of varying dimensions can be drawn such that all nodes at the same depth are vertically aligned, which we call *layered* drawings, or nodes can be placed vertically at at fixed distance from each other, which we call *non-layered* drawings. The difference between a layered and a non-layered drawing can be seen in Figure 3.2. Both types of drawings have their own merits: layered drawings make it easy to compare the depth of nodes, whereas non-layered drawings are vertically more compact.

Using a simple trick which we introduce later, non-layered drawings can also be used to show an attribute of each node in its vertical coordinate. An example of this is a family tree diagram where the vertical coordinate top coordinate of a node signifies the birth year of the corresponding person, as shown in Figure 3.3. An example in biology is a diagram which shows evolutionary relationships between biological species and the time in which each species came into existence. Another example is a cell division diagram where the vertical coordinate of each cell indicates the time when its parent cell divided.

A layered drawing of a tree can be found in $O(n)$, where n is the number of nodes in the tree, using the well-known Reingold-Tilford [Reingold and Tilford, 1981] algorithm. Various algorithms [Bloesch, 1993; Hasan et al., 2003; Miyadera et al., 1998; Stein and Benteler, 2007; Xiaohong and Jingwei, 2010] have been proposed for the non-layered case, but all of these either make simplifying assumptions or have not been proven to run in linear time.

In this chapter, we extend the Reingold-Tiford algorithm such that it also works for non-layered drawings. The original complexity proof of the algorithm for layered trees makes use of an invariant that does not hold for the non-layered case. We give an alternative proof that does not use this invariant, and show how it is adopted to


(a) Layered.

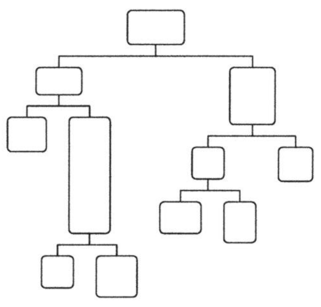

(b) Non-layered.

Figure 3.2: Layered and non-layered tidy drawings of the same tree.

prove that the extended Reingold-Tilford algorithm for the non-layered case also runs in $O(n)$.

To improve drawings of trees of unbounded degree, an extension to the Reingold-Tilford algorithm has been proposed [Walker, 1990; Buchheim et al., 2006]. This extension also applies to non-layered trees, but we show that they then cause a $O(n^2)$ run-time. We then present a modification to these techniques and prove that this modification restores the $O(n)$ run-time.

Our contributions can be summarized as follows:

- An extension of the Reingold-Tilford algorithm such that it can also produce non-layered drawings.
- A proof of the linear run-time of the Reingold-Tilford algorithm with this extension.
- A proof that the extension for trees of unbounded degree causes a $O(n^2)$ run-time in the non-layered case.
- A modification to this extension and proof that it restores $O(n)$ run-time.

The rest of this chapter is organized as follows; We first reformulate the tidy tree drawing problem to include non-layered drawings in Section 3.2. We then give an overview of the Reingold-Tilford algorithm and introduce its extension for non-layered drawings in Section 3.3. In Section 3.4, we prove that the extended Reingold-Tilford algorithm runs in linear time. We discuss the known extension (for layered trees) which improves the drawings of trees of unbounded degree in Section 3.5. Afterwards, we show that these techniques lead to a $O(n^2)$ run-time in the non-layered case and propose a modification to restore the $O(n)$ run-time in Section 3.6. We show measurements of the speed of this algorithm in Section 3.7. Finally, in Section 3.8 we discuss the history of this algorithm and related work. For sake of completeness, we discuss the techniques in the Reingold-Tilford algorithm which are also applicable

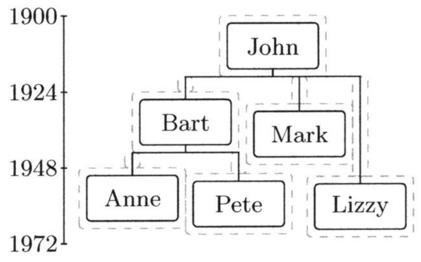


Figure 3.3: Descendants of John: layout with prescribed vertical positions (corresponding to birth year).

in the non-layered case in Appendix 3.A. In Appendix 3.B, we discuss the details of the parts of the techniques to improve drawings of unbounded degree which are also applicable in the non-layered case. In the Appendix 3.C we list the complete source code of the algorithm with the extensions discussed and proposed here.

3.2 Redefining the Tidy Tree Problem

In this section we reformulate the tidy tree drawing problem to include non-layered drawings. In order to reformulate cleanly, we abstract away from spacing between nodes and drawing connecting lines. The spacing between nodes is added by adding a gap to the widths and heights of the nodes. For example, the solid boxes in Figure 3.5(f) show the original widths and heights and the dashed boxes show the widths and heights after adding the gap.

Since we abstract away from spacing, in the layered setting, all nodes at the same depth can be considered to be of the same height. The input tree is then a rooted, ordered tree with a width for each node and a height for each depth in the tree. The vertical coordinate of each node is simply the vertical coordinate of its depth. In our more general non-layered setting, an input tree is also a rooted, ordered tree, but with a width and height for each node. Since we abstract away from spacing, the vertical top position of a node is then the bottom coordinate of its parent, which in turn is its top coordinate plus its height. In the rest of this chapter, we assume that the vertical positions of the input nodes have already been calculated in this way. Notice that if all the nodes at the same depth are of the same height, then a non-layered drawing is the same as a layered drawing.

Sometimes we have an input tree where the top coordinate of each child is not equal to the bottom coordinate of its parent, as is the case in the trees and in Figure 3.2(a) and Figure 3.3. We can transform such invalid trees to valid trees by adding thin "dummy" nodes between parent and child, as shown in Figure 3.4.

The tidy tree drawing problem is then reformulated as follows: given an input

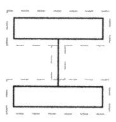


Figure 3.4: Dummy node

tree, produce a horizontal coordinate for each node such that drawing is compact[2] and the following aesthetic criteria are met [Reingold and Tilford, 1981; Buchheim et al., 2006]:

1. Nodes do not overlap.
2. Children are positioned horizontally in the order given in the tree.
3. Parents are centered above their children.
4. The drawing of a subtree does not depend on its position in the tree, i.e., identical subtrees are drawn identically.
5. The drawing of the reflection of a tree, i.e. the order of the children of each parent is reversed, is the mirror image of the drawing of the original tree.

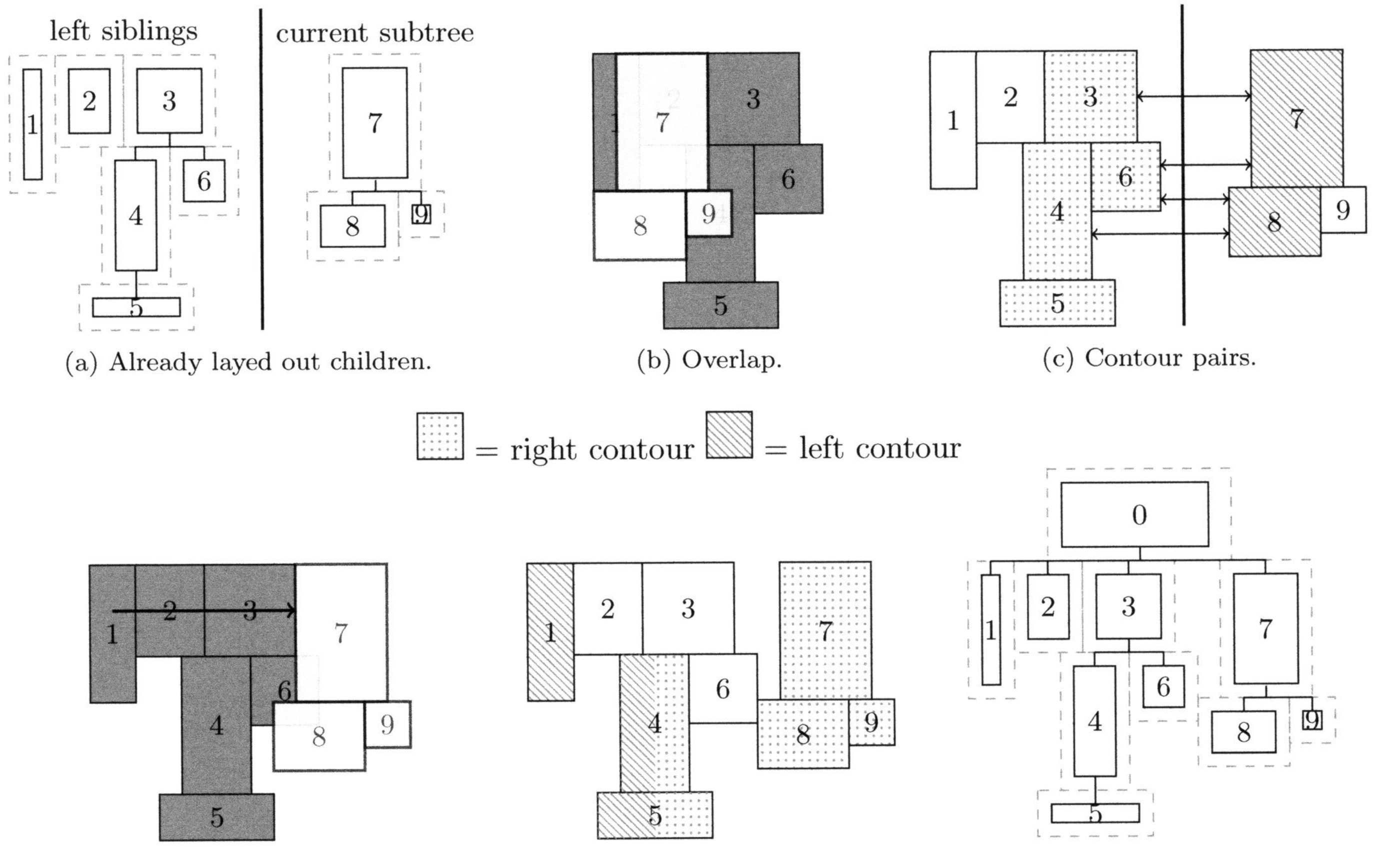
The rationale for these aesthetic criteria can be found in [Reingold and Tilford, 1981].

3.3 Overview of the extended Reingold-Tilford algorithm

We now give a high-level description of the Reingold-Tilford algorithm and introduce our extension for the non-layered case. A pseudo-algorithm for this high-level description is given in Algorithm 1. The Reingold-Tilford algorithm first recursively processes all the children of the tree, which produces a layout of each child as if it were the root, and hence the children overlap. Afterwards, each child subtree is moved to the right such that it does not overlap with its left siblings. After moving the children, the horizontal position of the root node is set such that it is centered above its children.

Moving the children works as follows: the algorithm iterates over the children from left to right, and moves each child subtree so that it does not overlap with any of its left siblings. Consider an example of such an iteration, shown in Figure 3.5. The start of the iteration is shown in Figure 3.5(a). Since the algorithm does not deal with spacing and connecting lines, we only show these at the start and at the end, i.e., in

[2]Compact meaning here that if we draw a vertical line from the top of the drawing to the bottom of the drawing at any horizontal coordinate inside the drawing, we will cross at least one node of the tree.



(a) Already layed out children.

(b) Overlap.

(c) Contour pairs.

(d) First move.

(e) Merged contours.

(f) Result.

Figure 3.5: Moving a child subtree.

```
Layout(root) begin
    foreach Each child of root do
        Layout(child);
        Separate((left siblings,child)) ;
    Set position of root;
Separate(left siblings, current subtree) begin
    /* The contour pair is the pair of these two variables.        */
    Current right contour node $\leftarrow$ root of rightmost sibling;
    Current left contour node $\leftarrow$ root of current subtree;
    while right contour node $\neq$ null $\wedge$ left contour node $\neq$ null do
        $x_l \leftarrow$ horizontal position of the left side of the current left contour node;
        $x_r \leftarrow$ horizontal position of the right side of the current right contour
        node;
        if $x_l < x_r$ then
            Move current subtree by $x_r - x_l$ to the right;
        $y_l \leftarrow$ vertical position of the bottom of the current left contour node;
        $y_r \leftarrow$ vertical position of the bottom of the current right contour node;
        /* Coordinate system increases upwards.                    */
        if $y_l \leqslant y_r$ then
            Current left contour node $\leftarrow$ next node of the left contour;
        if $y_l \geqslant y_r$ then
            Current right contour node $\leftarrow$ next node of the right contour;
    Merge contours;
```

Algorithm 1: High-level description of the extended Reingold-Tilford algorithm.

Figure 3.5(a) and Figure 3.5(f). In the left part of the Figure 3.5(a) we see that three left sibling subtrees were already layed out and moved, namely the subtree consisting of the node 1, the subtree consisting of the node 2 and the subtree consisting of the nodes $3, 4, 5, 6$. The subtree that should now be moved, from now on referred to as the *current* subtree, is shown in the right part of Figure 3.5(a) and consists of the nodes $7, 8, 9$. The layout of the current subtree and the left siblings is already correct, due to recursion. Notice that the left siblings and the current subtree are actually in the same space, and thus overlap as shown in Figure 3.5(b).

To see how much the current subtree must be moved, we use the *contours* of the current subtree and its left siblings. The left contour is the list of the nodes, from top to bottom, that can be "seen" from the left. The right contour is defined symmetrically. The contours in the example are shown in Figure 3.5(c). In our example, the left siblings have a left contour, consisting of the nodes $[1, 4, 5]$, and a right contour, consisting of the nodes $[3, 6, 4, 5]$. The current subtree has of a left contour, $[7, 8]$, and a right contour, $[7, 9, 8]$.

To move the current subtree, only the right contour of the left siblings and the left contour of the current subtree are needed. The algorithm then processes all *contour pairs*: vertically overlapping nodes from both contours. The contour pairs in our example are also shown in Figure 3.5(c).

In the layered case, finding the contour pairs works as follows: The first contour pair consists of the first node of the right contour and the first node of the left contour. The next contour pair is found by advancing both nodes from the current pair to the next element of their contour. We iterate this process until one of the contours has no more elements. In the non-layered case, the bottom coordinates of the contour nodes do not have to be equal. Hence, in each iteration only the highest one will be advanced to the next node of its contour, or both if they have the same bottom coordinates. This is the only modification needed to make the Reingold-Tilford algorithm work for non-layered trees. This modification consists of the two tests in lines 17 and 19 in our high-level description of the algorithm in Algorithm 1. In our non-layered example in Figure 3.5(c), this process yields contour pairs $\langle 3, 7\rangle, \langle 6, 7\rangle, \langle 6, 8\rangle$ and $\langle 4, 8\rangle$.

For each contour pair we then check if the left side of the left contour node is to the left of the right side of the right contour node. If this is the case, then the current subtree overlaps with its left siblings, and we move the current subtree such that the horizontal position of the left side of the left contour node is the same as the horizontal position of the right side of the node in the right contour. In our example the first contour pair is $\langle 3, 7\rangle$, and the left side of 7 is indeed to the left of the right side of 3, as can be seen in 3.5(b). We then move the current subtree to the right such that the left side of 7 is the right side of 3, as shown in Figure 3.5(d). The current subtree is then moved again for contour pairs $\langle 6, 7\rangle$ and $\langle 6, 8\rangle$, after which the current subtree is positioned as shown in Figure 3.5(e). Notice that this is *not* the same as simply moving the current subtree by the distance between the left of its leftmost node and the right of the rightmost node of its left sibling.

Afterwards the contours of the left siblings and the current subtree are merged into a new left and right contour, so that these are available later. In our example, the left siblings were taller than the current subtree. The left merged contour is then just the left contour of the left siblings, as shown in Figure 3.5(e). The merged right contour is the right contour of the current subtree, followed by the remainder of the right contour of the left siblings. More precisely, the merged left contour consists of the nodes $[1, 4, 5]$, and the merged right contour consists of the nodes $[7, 9, 8, 4, 5]$. After merging the contours the iteration ends and the algorithm starts moving the next child subtree. In our example, the subtree rooted at 7 was the last child, so the algorithm positions the root node using the positions of its children. The result can be seen in Figure 3.5(f).

A naïve implementation of the above algorithm will not run in linear time. In order to get a linear run-time the Reingold-Tilford algorithm uses techniques to do both of the following in $O(1)$:

- Getting the next element of a contour.
- Moving a subtree horizontally.

For the former, the algorithm maintains two fields called the left and right *threads* for each node, which contain a reference to the next node in the left or right contour respectively. Getting the next element of a contour is then simply using this reference. For the latter, the algorithm makes use of *relative coordinates.* The details of these techniques are given in Appendix 3.A. In the rest of this chapter, it suffices to know that these operations can be done in $O(1)$. It does not matter how this is achieved.

It should be clear that this algorithm satisfies aesthetic criteria 1-3, as listed on page 47. See [Gibbons, 1996] for a more in depth discussion of this. Aesthetic criteria 4 also holds, since the algorithm is a simple recursive algorithm that lays out each subtree in the same fashion, without taking into account where in the tree the subtree is located. Aesthetic criteria 5 does *not* hold, and we will see later how this can be fixed.

3.4 Complexity Proof

The original complexity proof of the Reingold-Tilford algorithm uses an invariant which does not hold in the non-layered case: the number of nodes in the left or right contour is equal to the depth of the tree, i.e. the length of the longest path from root to leaf. We will now give an alternative complexity proof, which does not use this invariant and then generalize this proof to the non-layered case.

The running time of the Reingold-Tilford algorithm depends on the total amount of contour pairs considered to move all subtrees. The total number of contour pairs is the same as the total number of times that the program executes the body of the while loop in Algorithm 1. The centering of a root above its children costs constant time per node, as we only need the positions of the leftmost and rightmost child. The moving of a subtree and the obtaining of the next node of a contour costs constant time per contour pair, by using the techniques explained in Appendix 3.A. Hence, if the total number of contour pairs processed during the layout of the entire tree is linear in the size of the tree, then the Reingold-Tilford algorithm runs in linear time.

3.4.1 Layered case

Let us first assume that the input tree is *layered.* The key insight for this complexity proof is the following: if a node in the left contour of a subtree was processed to move that subtree, then it *cannot* be part of the left contour of another subtree. As an example, consider Figure 3.1. Here the left contour of the right child consisted of $4, 5$ and 6. The left contour nodes considered when moving the right child are 4 and 5, which thus *cannot* reoccur in another left contour. The reason for this is that after moving the current subtree, the left contour nodes that were processed will all have a node in the left siblings to the left of them. In our example, the processed left contour nodes 4 and 5 have the nodes 1 and 3 to the left of the them respectively. In other words, since the left contour is all the nodes that can be "seen" from the left, the processed left contour nodes of the current subtree cannot be part of the merged contour, because they are occluded by nodes in the left siblings. An analogous insight holds for the right contour nodes.

More formally, the input tree consists of n nodes, $v_1 \ldots v_n$. Let $f_l(v_i)$ be the set of left contour nodes processed to move the subtree with root v_i. Due to the above insight, we know that if a node v_i is in a set $f_l(v_j)$, it cannot be a part of any other set $f_l(v_z)$, with $z \neq j$. Because of this, we know that the total number of left contour nodes processed to layout the entire tree is less than or equal to n, i.e.:

$$\sum_{i=1}^{n} |f_l(v_i)| \leqslant n$$

Where $|x|$ denotes the number of elements in the set x. The equivalent holds for $f_r(v_i)$, the set of right contour nodes that where processed to move the subtree with root v_i.

Let $f(v_i)$ be the set of contour pairs processed to move the subtree with root v_i. In the layered case, nodes are aligned vertically, and hence we know that $|f_l(v_i)| = |f_r(v_i)| = |f(v_i)|$, where $|x|$ denotes the size of the set x. Hence, the amount of contour pairs processed during the entire algorithm is less than or equal to n, which means that the Reingold-Tilford algorithm runs in linear time.

3.4.2 Non-layered case

In the non-layered case, nodes are not necessarily vertically aligned. Hence, the amount of contour pairs processed to move a child is no longer the same as the number of left contour elements processed. In the worst case the nodes from the right and left contours are never aligned, i.e., their bottom coordinates are never the same. After processing a contour pair, we will advance either along the left or the right contour or along both. This gives us an upper bound on the number of contour pairs:

$$|f(v_i)| \leqslant |f_l(v_i)| + |f_r(v_i)|$$

Another difference to the layered case is that a node in a left contour now *can* be processed to move the subtree *as well as* being included in another left contour.

Again, the same holds for right contours. As an example of a right contour node that is also included in another right contour, consider the tree in Figure 3.5(c). In this example, right contour node 4 is processed when moving the subtree rooted at 7 and it is also in the right contour of the entire tree, rooted at 0, as shown in Figure 3.5(d).

However, this can *only* happen if the node is the *last* right contour node that was processed to move a subtree. The reason for this is that the top part of the last node that was considered is occluded by nodes to the right while other nodes that were considered must be *totally* occluded by nodes to the right. Again, as an example, consider Figure 3.5(c). The last considered right contour node of the left siblings, node 4, is partially occluded by the nodes 7 and 8 to the right in the merged contour in Figure 3.5(d). However the other right contour nodes that were considered , namely 3 and 6, are totally occluded by nodes to the right, namely 7 and 8. The same reasoning holds for the last left contour node that was processed to move a subtree.

Let $f_l^p(v_i)$ be the set of left contour nodes that where processed to move the subtree with root v_i, *except* the last left contour node that was processed. More formally $f_l^p(v_i) = f_l(v_i) - \{l_l(v_i)\}$, where $l_l(v_i)$ is the last node of the left contour that is processed to move the subtree with root v_i. Since only last elements of a contour can reappear, we know that:

$$\sum_{i=1}^{n} |f_l^p(v_i)| \leqslant n$$

When moving each subtree, there will be only one last left contour element considered, and since there are n subtrees, we know that:

$$\sum_{i=1}^{n} |f_l(v_i)| = \sum_{i=1}^{n} [|f_l^p(v_i)| + |\{l_l(v_i)\}|] = \sum_{i=1}^{n} |f_l^p(v_i)| + n \leqslant 2n$$

The same argument can be made for the right contour, and hence we know that:

$$\sum_{i=1}^{n} |f(v_i)| \leqslant \sum_{i=1}^{n} |f_l(v_i)| + \sum_{i=1}^{n} |f_r(v_i)| \leqslant 4n = O(n)$$

Which proves that the extension of the Reingold-Tilford algorithm for non-layered trees also runs in linear time.

3.5 Improving layouts

The above Reingold-Tilford algorithm for non-layered trees satisfies aesthetics 1-4, but not aesthetic 5 [Walker, 1990]: the drawing of the reflection of a tree is not the mirror image of the drawing of the original tree. An example of this is shown in Figure 3.6(a). When subtrees are enclosed by larger siblings they will be piled to the left. If we draw the reflected tree and then mirror the layout, the subtrees are piled to the right as shown in Figure 3.6(b). A simple trick to satisfy aesthetic 5 is to take

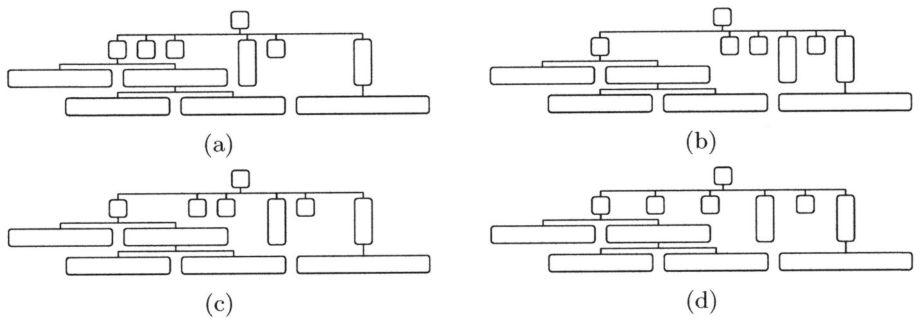


Figure 3.6: Layouts not satisfying aesthetic 5 (a,b) and satisfying aesthetic 5(c,d). Inspired by a figure in [Buchheim et al., 2006].

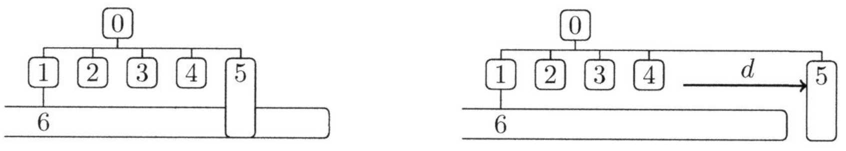
the average of the horizontal position of the each node in the original and mirrored, reflected drawing, which results in a layout as shown in Figure 3.6(c). However, this tends to cluster smaller subtrees, which is less aesthetically pleasing. Walker [Walker, 1990] noticed this problem and proposed extensions to the Reingold-Tilford algorithm to produce aesthetically more pleasing layouts, such as the one shown in Figure 3.6(d). Buchheim, Jünger and Leipert [Buchheim et al., 2006] then showed that the Walker algorithm runs in $O(n^2)$ and provided techniques to restore the linear running time.

To see how such layouts are achieved, consider the example shown in Figure 3.7(a). Here we see that we have already layed out and moved children 1-4 and we are currently moving child 5 to the right. We already processed the first node of the right contour of the left siblings, namely 4, and hence the left side of node 5 is no longer to the left of the right side of node 4. We now move on to the next node of the right contour, 6. As shown in Figure 3.7(b), we need to move node 5 by a distance d to the right. In the Reingold-Tilford algorithm as described before this would have been the only thing we would have done. To satisfy aesthetic 5, we notice that the current node in the right contour, 6, which caused the move by d, is in the sibling subtree with root 1.

If we move 5 by d, then there is d space between 5 and its left sibling, 4, as shown in Figure 3.7(b). We can then distribute this extra space over the gaps between the intermediate siblings, the siblings between the sibling that caused the move and the current subtree, namely 1 through 4. Since there are 4 gaps, we move the first intermediate sibling node by a distance $\frac{1}{4}d$, the second by $\frac{2}{4}d$, and the third by $\frac{3}{4}d$, as shown in Figure 3.7(c).

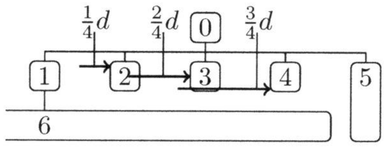
In general, suppose a node in the current subtree is a distance d to the left of a node v in the right contour of its left siblings. After moving the current subtree by d, we then see which left sibling is the ancestor of v. Let i be the index of this left sibling and j be the index of the current subtree. We then move each intermediate sibling, with an index z the range $[i+1 \ldots j-1]$, by a distance $\frac{z-i}{j-i}d$.

To do the above modification to the Reingold-Tilford algorithm while retaining the running time of $O(n)$, Buchheim et al. introduce techniques to do both of the



(a) Before moving 5.

(b) After moving 5.



(c) Distributing the space over intermediate siblings.

Figure 3.7: Shifting intermediate children

following in $O(1)$:

- Move the intermediate siblings as described above.
- Given a node in the right contour, get the index of the sibling subtree which contains that node.

For the first point, Buchheim et al. propose a technique which is also applicable in the non-layered case. Its details are not important in this chapter, but it is explained in Appendix 3.B for sake of completeness.

For the second point, the Buchheim et al. propose a technique which requires updating all the nodes in the right contour of a subtree after moving that subtree, but only if the subtree is less tall than its left siblings. If a subtree is less tall than its left siblings, then all its left contour nodes are considered to move that subtree. In the layered case, the left and right contours must have exactly as many elements. Therefore, the number of right contour nodes of a subtree that is less tall than its left siblings is the same as the number of contour pairs considered to move that subtree. Hence, updating the right contour of subtrees that are less tall than its left siblings does not modify the $O(n)$ run-time in the layered case. In the non-layered case this technique causes a run-time of $O(n^2)$, which we will show and remedy in the next section.

3.6 Improving layouts in linear time

In the non-layered case, the left and right contours do not have to have the same number of elements. Because of this, updating the right contour nodes of subtrees that are less tall then their left siblings leads to an $O(n^2)$ run-time. As an example of this, consider the tree construct shown in Figure 3.8. Formally, we construct a tree given a parameter k: the root node has width and height 2^k, and has three children:

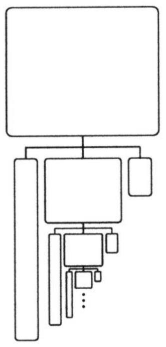


Figure 3.8: A tree construct where the sibling index lookup technique of Buchheim et al. gives $O(n^2)$ run-time.

- A child consisting of a single node of width $\frac{1}{4}2^k$ and height $\frac{5}{4}2^k$.
- A child subtree constructed in the same way, with $k = k - 1$.
- A child consisting of a single node of width $\frac{1}{4}2^k$ and height $\frac{1}{4}2^k$.

When $k = 1$, the tree is constructed in the same way, but the middle child is instead a single node with width and height 1.

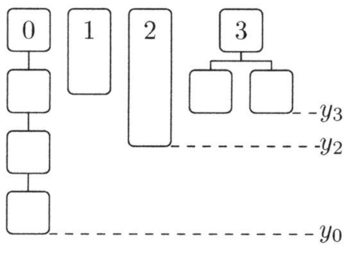
For every k, the middle child subtree is less tall then its left sibling. Hence, when using Buchheim et al.'s technique we must always update the nodes in the right contour of the middle child after moving it to the right. For each k, a tree constructed in this way has $3k + 1$ nodes. The right contour of the middle child consists of all nodes in its subtree, i.e. $3(k-1)+1$ nodes. Updating the right contour *for all* middle children in in such a tree then takes:

$$\sum_{i=1}^{k-1}[3i+1] = 3\sum_{i=1}^{k-1} i + k - 1 = O(k^2)$$

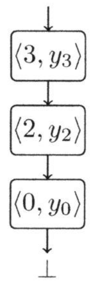
Due to the well-know equality $\sum_{i=1}^{k} i = k(k+1)/2$. Since there is a linear relation between n and k, this means that the algorithm runs in $O(n^2)$.

Hence, we need a different technique to find the index of the sibling subtree that contains a given node in the right contour. As noted by Buchheim et al., it is also possible to adopt the lowest common ancestor algorithm of Schrieber and Vishkin [Schieber and Vishkin, 1988] to find the index of the sibling subtree in $O(1)$, after an $O(n)$ preprocessing step. This is indeed possible, but not trivial, and Buchheim et al. do not describe the details.

We propose a different, much simpler technique: during the moving of the children we maintain a linked list of the siblings that currently have a node in the right contour. Each node in this linked list is a pair of the index of the corresponding sibling and its lowest vertical bottom coordinate. This list is always sorted in descending order



(a) Left siblings and their lowest vertical coordinates.



(b) Left sibling lookup list.

Figure 3.9: A tree layout with the corresponding linked list.

of the siblings indices. An example of a tree with the corresponding list is shown in Figure 3.9. When moving a child subtree we advance this list if the current right contour node has a lower vertical coordinate than the pair at the head of the list, adding only $O(1)$ operations per contour pair. The index of the sibling subtree that contains the current right contour node is then always given in the pair at the head of the list.

After moving a current subtree to the right, we need the pair for the current subtree, i.e. the subtree that was just moved. This pair consists of the sibling index of the current subtree, which we already know, and the lowest vertical bottom coordinate of the current subtree. The latter is easily found when using the techniques in the Reingold-Tilford algorithm as described in Appendix 3.A. More precisely, the Reingold-Tilford algorithm keeps track of the *extreme nodes* of each subtree, i.e. the lowest nodes that can be "seen" from the left or right. The lowest vertical bottom coordinate of a subtree is then the bottom coordinate of either of its extreme nodes, and can hence be found in $O(1)$.

To update the list, we then remove elements at the head of the list that have a higher lowest vertical coordinate than the new pair. This removes siblings from the list that had nodes in the right contour, but these nodes are now occluded by the current subtree. Afterwards, we prepend the new pair to the list. In this way, the list always corresponds to the siblings that currently have a node in the right contour.

The total number of operations needed for updating the list when moving all the children of a node is at most $2m(v_i)$, where $m(v_i)$ is the number of children of a node v_i. This can be seen as follows: we update the list $m(v_i)$ times during the moving of the children, namely after moving each child. Each time we remove some number of elements from the head of the list and an element is prepended to the list. Since we add $m(v_i)$ elements to the list, and an element can only be removed once, the total number of elements dropped is at most $m(v_i)$. Hence, together the prepending and dropping of elements cost at most $2m(v_i)$ operations per node.

For the entire tree this will add at most $\sum_{i=1}^{n} 2m(v_i)$ operations. Every node is a child of exactly one other node, except for the root who is a child of no one. Hence,

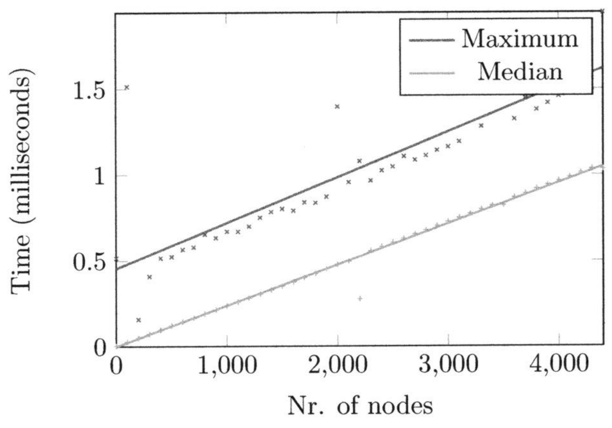


Figure 3.10: Measurements on random trees. The minimum is not shown, as it coincides with the x-axis.

the sum of the number of children of all nodes, $\sum_{i=1}^{n} m(v_i)$, is equal to $n-1$. Because of this we know that the updating of the lists cost

$$\sum_{i=1}^{n} 2m(v_i) = 2\sum_{i=1}^{n} m(v_i) = 2(n-1) = O(n)$$

extra operations for the entire tree. Since the updating of the place in the list during the moving of a subtree cost $O(1)$ per contour pair, this will also add $O(n)$ extra operation to the algorithm. Hence, the running time of the algorithm with this extension is linear in the size of the tree.

3.7 Empirical results

In practice the algorithm presented in this chapter is very fast, and should hence be applicable in any computing environment. In Figure 3.10 we show the results of measuring the time the algorithm took to lay out randomly generated trees. We used the following procedure to generate a tree with n nodes: Start with a tree with a single node. Then, for each other node, add it as a leaf in the current tree in a random position. This random position is determined by descending the current tree as follows: Starting with the root node as the current node, we generate a random integer between zero and the number of children of the current node. If the random number is zero, then new node is added as a new child of the current node. Otherwise, we descend into the child with as index the random number and repeat the procedure. Each node has a random width and height uniformly distributed between 1.0 and 10.0. As there are more possible trees as the number of nodes goes up, we generated $200 \times n$ tests, where n is the number of nodes, for each multiple of 100 in the range [0,4400]. These results were obtained using OpenJDK java runtime version 2.3.9.8 on an Intel i7 2.8 GHz CPU running Fedora Linux 3.9.4-200.fc18.x86_64. This experiment can be repeated by downloading the source code from `http://github.com/cwi-swat/non-layered-tidy-trees`.

3.8 Related work

3.8.1 History

The history of the Reingold-Tilford algorithm is quite long: In 1979, Wetherell and Shannon [Wetherell and Shannon, 1979] presented the first $O(n)$ algorithm that produces drawing satisfying aesthetics 1-3, that was inspired by a tree drawing algorithm presented by Knuth in 1971 [Knuth, 1971]. Two years later, Reingold and Tilford [Reingold and Tilford, 1981] gave an algorithm, inspired by the Wetherell and Shannon algorithm, that also satisfied aesthetic 4. Then in 1990, Walker [Walker, 1990] presented an improvement such that aesthetic 5 is also satisfied for trees of unbounded degree. In 2002 Buchheim, Jünger and Leipert showed that Walker's algorithm ran in $O(n^2)$, in contrast to what the author claimed. They presented improvements to the Reingold and Tilford algorithm inspired by Walker's work that did run in $O(n)$. All these versions of the Reingold-Tilford algorithm, and their proofs assume layered trees.

3.8.2 Algorithms for non-layered trees

There have been several previous efforts to produce non-layered drawings of trees. All of these either make simplifying assumptions and/or have not been proven to linear time.

Miyadera et al. present an $O(n^2)$ algorithm [Miyadera et al., 1998] for non-layered trees that horizontally positions a parent at a fixed offset from its first child, instead of centered above the children. This greatly simplifies things, as only the first child needs to be layed out before positioning the root node, allowing a simple depth-first solution. A proof that such a $O(n)$ depth-first solution exists was given by Hasan and Radwan [Hasan et al., 2003]. This type of layout can also be easily handled by the extended Reingold-Tilford algorithm, by simply modifying the computation of the position of the root node. This will break aesthetic 5, as the Hasan and Miyadera algorithms also do.

Bloesch gives two algorithms [Bloesch, 1993] for non-layered trees that are intended to satisfy aesthetics 1-5. Both algorithms follow the same idea: discretize the drawing vertically. The first is then a variant of an algorithm for layered trees by Vaucher [Vaucher, 1980] and the second is a variant of the original Reingold-Tilford algorithm. This variant does not use threads (described in Appendix 3.A) like the original Reingold-Tilford algorithm, as the author states that this is impossible in a non-layered setting, which is obviously false. It is unclear to us how the drawing generated by these algorithms satisfy aesthetics 4 and 5. Bloesch reports that these algorithms run in $O(nh)$, where h is the number of elements in the discretization of the height. It is unclear to us whether this is true, as no proof is given.

Stein and Benteler [Stein and Benteler, 2007] propose a similar technique: A non-layered tree is converted into a layered tree by discretizing horizontally and vertically. Afterwards, an algorithm for layered trees can be applied and the results can be translated back. This approach runs in $O(f(n)wh)$, where w and h are the number of elements in the horizontal and vertical discretization respectively and $f(n)$ is the running time of the algorithm for layered trees.

Xiaohong and Jingwei [Xiaohong and Jingwei, 2010] present an algorithm for non-layered trees satisfying aesthetic criteria 1-4. The algorithm is presented as a complete novelty, but it is the Reingold-Tilford algorithm with the small extension that we introduced in Section 3.3, which amounts to three extra lines in the algorithm as shown in Appendix 3.C. In contrast to our work, no proof is given of the time complexity nor do the layouts by algorithm satisfy aesthetic criteria 5.

Marriot, Sbarski, Van Gelder, Prager and Bulka [Marriott et al., 2011] presented, among other things, a technique to produce a more vertically compact drawing of a tree where the nodes have different heights. This technique works by first pre-processing the tree and then using the Reingold-Tilford algorithm extended with the techniques of Walker and Buchheim. In the pre-processing step, the tree is "re-layered": if the distance between a parent and a child is too large, the layer is split in two. In this way, a more vertically compact layout can be obtained in $O(n \log n)$. With the algorithm presented in this chapter we can achieve a drawing with *minimal* height in $O(n)$. A non-layered drawing has minimal height, since the top-coordinate of a node in the bottom coordinate of its parent (abstracting away from spacing).

3.8.3 Related work on node-link drawings on trees

Apart from algorithms for drawing node-link diagrams of trees, various results have been published on other aspects of node-link diagrams of trees: Gibbons [Gibbons, 1996] derives the Reingold-Tilford algorithm for binary trees from the aesthetic criteria. His derivation of the algorithm shows that the Reingold-Tilford is the only reasonable algorithm that satisfies the aesthetic criteria.

Kennedy [Kennedy, 1996] shows how to implement a variant of the Reingold-Tilford algorithm in a purely functional setting, with time complexity $O(n^2)$. We note that it should be possible to implement the Reingold-Tilford algorithm with its extensions in a purely functional setting while retaining the $O(n)$ run-time. This could work by separating the contours from the tree itself, i.e. maintaining a separate list representing the contour instead of reusing the tree structure for this. This would get rid of the need for mutability. If we then also choose a purely functional data structure for such a contour list with $O(1)$ first and last element access and $O(1)$ list concatenation, such as the data structure given by Kaplan and Tarjan [Kaplan and Tarjan, 1999a], we can implement a purely functional version running in $O(n)$.

Suppowit and Reingold [Supowit and Reingold, 1983] investigated the complexity of drawing trees nicely. They found that a drawing with global minimum width may have subtrees that are much wider than necessary. For this reason the Reingold-Tilford algorithm does not promise minimal width. They also found that the tidy tree problem can be reduced to a linear programming problem and that it is NP-hard if the horizontal coordinates are restricted to integers.

Moen [Moen, 1990] shows a variant of the Reingold-Tilford algorithm that works in approximately the same way, the main difference being that he uses a separate data structure for the contour instead of reusing the tree itself. He then shows how to keep the layout of the tree up-to-date when there are insertions and deletions in the tree.

Marriot and Sbarski [Marriott and Sbarski, 2007] relax the requirement that a parent must be placed exactly between the children, making it a preference that may be violated if it yields a tree with a smaller width. Their approach is to first find a initial layout using the (extended) Reingold-Tilford algorithm and then solve a kind of quadratic programming problem to see where the preference should be violated to produce a more narrow drawing. This technique requires that the drawing is divided into layers. However, in another publication [Marriott et al., 2011], Marriot and Sbarski showed how to this technique can be applied in a setting where a node can span *multiple* layers. Hence, if we introduce a layer for each unique vertical position and then assign nodes to layers, then this technique can also be used together with the algorithm presented in this chapter.

3.8.4 Other ways of drawing trees

We will now give a short and by no means complete overview of other tree visualization methods, for a more complete overview see [Katifori et al., 2007; Nguyen and Huang, 2002].

There are many variations on the basic node-link diagram. One way to adapt node-link diagrams is to change the coordinate system in which they are drawn. Radial trees draw node-link diagrams in a polar coordinate system, where the root is displayed at the origin. Balloon trees are similar, but the children of *each* node are in a circle around the node instead. The hyperbolic browser[Lamping et al., 1995] also changes the coordinate system in which a node-link diagram is draw, namely to a hyperbolic plane.

Another way to adapt node-link diagrams is to move from 2D to 3D. Cone trees [Robertson et al., 1991] are a 3D generalization of balloon trees: viewed from the top the diagram is a balloon tree, while viewed from the side we see a cone from each root node to its children. Another 3D generalization of node-link diagrams is visualize a hierarchy as as a botanical tree [Kleiberg et al., 2001].

Instead of node-link diagram, the parent-child relationship can also be visualized by *containment*: the children are drawn inside the parent. Treemaps [Johnson and Shneiderman, 1991] draw nodes as rectangles, and each rectangle is subdivided into the rectangles of the children. The area of each rectangle signifies the size of the node, for example the size of a file or the total size of a directory when visualizing a file system.

Hi-trees [Marriott et al., 2011] combine containment and node-link diagrams: the parent-child relationship is visualized by either a link or containment, depending on the type of the relationship between parent and child. For example, arguments in a discussion can have sub-arguments (containment) and supporting and opposing arguments (links).

Acknowledgements

We thank Leen Torenvliet, Menno van der Ploeg, Anastasia Izmaylova, Paul Klint and the anonymous reviewers for their constructive comments on this chapter.

Appendices to Chapter 3

3.A Techniques in the Reingold-Tilford algorithm

In order to get a linear run-time the Reingold-Tilford algorithm [Reingold and Tilford, 1981] uses techniques to do both of the following in $O(1)$:

- Getting the next element of a contour.
- Moving a subtree horizontally.

The operation for the first item, getting the next node of a contour, depends on whether the current node is a leaf or not. If the node is not a leaf, then the next element of the left contour is its leftmost child, and the next element of the right contour is its rightmost child. For leafs, the next element of the left and right contours are stored in two fields of each leaf, called the left and right *threads*. To keep the threads up-to-date, the algorithm has two additional fields per node: the left and right *extreme nodes*. The left and right extreme nodes of a set of siblings is the lowest node in the subtree the can be "seen" from the left and right respectively. For an example of threads and extreme nodes, see Figure 3.11(a). Before moving the current subtree, its left and right extreme nodes point to the extreme nodes in the current subtree. After moving the current subtree, its right extreme node points to the extreme node of the current subtree *and* its left siblings. The left extreme node of the first sibling always points to the left extreme node of the siblings that are already moved. Thus, the left extreme of a set of siblings that is already moved is a field of the leftmost sibling, and the right extreme is a field of the rightmost sibling.

After moving a current subtree, the threads and extreme nodes may be updated. If the current subtree was less tall than its left siblings, as is the case in Figure 3.11, the right thread of the extreme right node of the current subtree is set. Afterwards, the extreme right node of the root of the current subtree is set to the extreme right node of its left siblings. The resulting situation in our example is shown in Figure 3.11(b). If the current subtree was taller than its left siblings, the operation is symmetrical. If the current subtree is as tall as its left siblings no threads are set and no extreme nodes are updated.

To achieve moving a subtree horizontally in $O(1)$, the Reingold-Tilford algorithm uses a field named *mod*, for modifier, to store for each node how much its *entire* subtree should be moved horizontally. Moving a subtree is then done by simply updating this modifier. Another field, *prelim* is used to remember the *preliminary* horizontal coordinate of the node. This is set when we position the root after moving its children and represents the distance that the left side of the root is positioned relative to the left side of its first child. After laying out the entire tree, a single extra pass over the tree to computes the actual horizontal coordinate of each node. This use of relative coordinates requires some changes to the rest of the algorithm: during the moving of a subtree, we must maintain and take into account the sum of modifiers

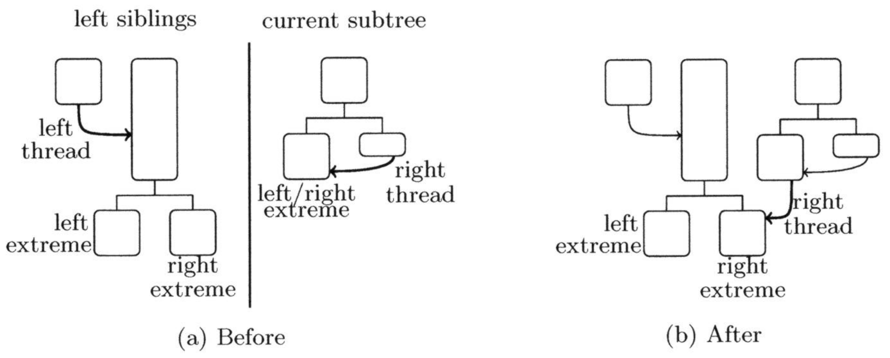


Figure 3.11: Before and after settings threads and updating extreme nodes.

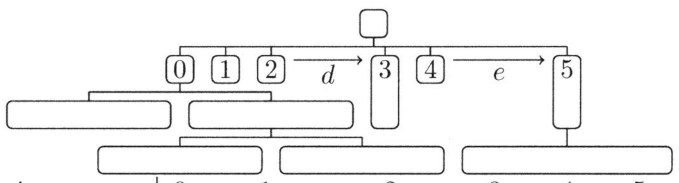
along the left and right contours to compute the horizontal positions of the contour nodes.

When setting a thread, we must ensure that if we follow the thread to a node, the sum of the modifiers along the contour is the same as the sum of the modifiers along the route without threads from the root to that node. We will only set threads of extreme nodes, which must be leafs. Hence if we adjust the modifier of an extreme node and adjust its preliminary horizontal position by an opposite amount, we will not actually change the position of any node. In this way we can adjust the modifier of the extreme node such that the sum of modifiers after following the thread is equal to the sum of modifiers when following the route without threads.

3.B Moving intermediate siblings in $O(1)$

To move an arbitrary number of intermediate siblings in $O(1)$, Buchheim et al. [Buchheim et al., 2006] propose to add two fields to each node, namely *shift* and *change*. Suppose, like in Section 3.5, that i is the index of the sibling which is an ancestor of the current node in the right contour, j is the index of the current subtree, and we move the current child by a distance d. We then add $\frac{1}{j-i}d$ to the *shift* of the $i+1$th child, subtract $\frac{1}{j-i}d$ from *shift* of jth child and subtract $\frac{j-i-1}{j-i}d$ from the *change* of the jth child, as shown in the method `distributeExtra` in Appendix 3.C. Together, these operations cost only constant time.

After laying out the entire tree, we do a constant number of extra operations per node to compute the actual offset of each child, using these *shift* and *change* fields. This can be done during the post-processing phase that produces the absolute horizontal coordinates. Before descending into the children, the *shift* and *change* fields are used to calculate the change to *mod*, Δmod, to each child, as shown in method `addChildSpacing` in Appendix 3.C. An example is shown in Figure 3.12. The example shows the tree from Figure 3.6, and adding Δmod to the *mod* of each child will transform 3.6(a) to 3.6(d).



i	0	1	2	3	4	5
$shift(T_i)$	0	$\frac{1}{3}d + \frac{1}{5}e$	0	$-\frac{1}{3}d$	0	$-\frac{1}{5}e$
$change(T_i)$	0	0	0	$-\frac{2}{3}d$	0	$-\frac{4}{5}e$
d	0	$\frac{1}{3}d + \frac{1}{5}e$	$\frac{1}{3}d + \frac{1}{5}e$	$\frac{1}{5}e$	$\frac{1}{5}e$	0
Δmod	0	$\frac{1}{3}d + \frac{1}{5}e$	$\frac{2}{3}d + \frac{2}{5}e$	$\frac{3}{5}e$	$\frac{4}{5}e$	0

Figure 3.12: Example of the post-processing of the spacing of intermediate children.

3.C The complete revised algorithm

Below is the full Java code for the Reingold-Tilford algorithm with its extension. This implementation has undergone rigorous testing, including checking for overlapping nodes on random input trees. The extensions are indicated by a symbol in the right margin.

(nothing)	=	The original Reingold-Tilford algorithm (Section 3.3 and Appendix 3.A).
$\star$	=	Our extension for non-layered trees (Section 3.3).

Extensions for satifying aesthetic 5 (Section 3.5)

$\circ$	=	Buchheim et al.'s extension to move intermediate siblings (Appendix 3.B).
$\bullet$	=	Our extension to look up the sibling index of a right contour node (Section 3.6).

```
class Tree {
   double w, h;            // Width and height.
   double x, y, prelim, mod, shift, change;
   Tree tl, tr;            // Left and right thread.
   Tree el, er;            // Extreme left and right nodes.
   double msel, mser;      // Sum of modifiers at the extreme nodes.
   Tree[] c; int cs;       // Array of children and number of children.

   Tree(double w, double h, double y,Tree... c) {
      this.w = w; this.h = h; this.y = y; this.c = c;
      this.cs = c.length;
   }
 }
void layout(Tree t){ firstWalk(t); secondWalk(t,0); }

void firstWalk(Tree t){
  if(t.cs == 0){ setExtremes(t); return; }
  firstWalk(t.c[0]);
  // Create siblings in contour minimal vertical coordinate and index list.
  IYL ih =  updateIYL(bottom(t.c[0].el),0,null);                          •
  for(int i = 1; i < t.cs; i++){
     firstWalk(t.c[i]);
     //Store lowest vertical coordinate while extreme
     //nodes still point in current subtree.
     double minY = bottom(t.c[i].er);                                      •
     separate(t,i,ih);
     ih = updateIYL(minY,i,ih);                                            •
  }
  positionRoot(t);
  setExtremes(t);
}

void setExtremes(Tree t) {
   if(t.cs == 0){
      t.el = t; t.er = t;
      t.msel = t.mser =0;
   } else {
      t.el = t.c[0].el; t.msel = t.c[0].msel;
      t.er = t.c[t.cs-1].er; t.mser = t.c[t.cs-1].mser;
   }
}

void separate(Tree t,int i,  IYL ih ){
   // Right contour node of left siblings and its sum of modfiers.
   Tree sr = t.c[i-1]; double mssr = sr.mod;
   // Left contour node of current subtree and its sum of modfiers.
   Tree cl = t.c[i]  ; double mscl = cl.mod;
   while(sr != null && cl != null){
      if(bottom(sr) > ih.lowY) ih = ih.nxt;                                •
      // How far to the left of the right side of sr is the left side of cl?
      double dist = (mssr + sr.prelim + sr.w) - (mscl + cl.prelim);
      if(dist > 0){
         mscl+=dist;
         moveSubtree(t,i,ih.index,dist);
      }
      double sy = bottom(sr), cy = bottom(cl);
      // Advance highest node(s) and sum(s) of modifiers
      // (Coordinate system increases downwards)
      if(sy <= cy){                                                        ★
         sr = nextRightContour(sr);
         if(sr!=null) mssr+=sr.mod;
      }                                                                    ★
      if(sy >= cy){                                                        ★
         cl = nextLeftContour(cl);
         if(cl!=null) mscl+=cl.mod;
      }                                                                    ★
   }
   // Set threads and update extreme nodes.
   // In the first case, the current subtree must be taller than the left siblings.
   if(sr == null && cl != null) setLeftThread(t,i,cl, mscl);
   // In this case, the left siblings must be taller than the current subtree.
  else if(sr != null && cl == null) setRightThread(t,i,sr,mssr);
}

void moveSubtree(Tree t, int i, int si, double dist) {
  // Move subtree by changing mod.
  t.c[i].mod+=dist; t.c[i].msel+=dist; t.c[i].mser+=dist;
  distributeExtra(t, i, si, dist);                              ○
}

Tree nextLeftContour(Tree t) {return t.cs==0 ? t.tl : t.c[0];}
Tree nextRightContour(Tree t){return t.cs==0 ? t.tr : t.c[t.cs-1];}
double bottom(Tree t) { return t.y + t.h;  }

void setLeftThread(Tree t, int i, Tree cl, double modsumcl) {
  Tree li = t.c[0].el;
  li.tl = cl;
  // Change mod so that the sum of modifier after following thread is correct.
  double diff = (modsumcl - cl.mod) - t.c[0].msel ;
  li.mod += diff;
  // Change preliminary x coordinate so that the node does not move.
  li.prelim-=diff;
  // Update extreme node and its sum of modifiers.
  t.c[0].el = t.c[i].el; t.c[0].msel = t.c[i].msel;
}

// Symmetrical to setLeftThread.
void setRightThread(Tree t, int i, Tree sr, double modsumsr) {
  Tree ri = t.c[i].er;
  ri.tr = sr;
  double diff = (modsumsr - sr.mod) - t.c[i].mser ;
  ri.mod += diff;
  ri.prelim-=diff;
  t.c[i].er = t.c[i-1].er; t.c[i].mser = t.c[i-1].mser;
}

void positionRoot(Tree t) {
  // Position root between children, taking into account their mod.
  t.prelim = (t.c[0].prelim + t.c[0].mod + t.c[t.cs-1].mod +
              t.c[t.cs-1].prelim +  t.c[t.cs-1].w)/2 - t.w/2;
}

void secondWalk(Tree t, double modsum) {
  modsum+=t.mod;
  // Set absolute (non-relative) horizontal coordinate.
  t.x = t.prelim + modsum;
  addChildSpacing(t);                                           ○
  for(int i = 0 ; i < t.cs ; i++) secondWalk(t.c[i],modsum);
}

void distributeExtra(Tree t, int i, int si, double dist) {      ○
  // Are there intermediate children?
  if(si != i-1){                                                ○
    double nr = i - si;                                         ○
    t.c[si +1].shift+=dist/nr;                                  ○
    t.c[i].shift-=dist/nr;                                      ○
    t.c[i].change-=dist - dist/nr;                              ○
    }                                                                   ○
}                                                                       ○

// Process change and shift to add intermediate spacing to mod.
void addChildSpacing(Tree t){
    double d = 0, modsumdelta = 0;                                      ○
    for(int i = 0 ; i < t.cs ; i++){                                    ○
        d+=t.c[i].shift;                                                ○
        modsumdelta+=d + t.c[i].change;                                 ○
        t.c[i].mod+=modsumdelta;                                        ○
    }                                                                   ○
}                                                                       ○

// A linked list of the indexes of left siblings and their lowest vertical coordinate.
class IYL{                                                              ●
    double lowY; int index; IYL nxt;                                    ●
    public IYL(double lowY, int index, IYL nxt) {                       ●
            this.lowY = lowY; this.index = index; this.nxt = nxt;       ●
    }                                                                   ●
 }                                                                      ●

IYL updateIYL(double minY, int i, IYL ih) {                             ●
    // Remove siblings that are hidden by the new subtree.
    while(ih != null && minY >= ih.lowY) ih = ih.nxt;                   ●
    // Prepend the new subtree.
    return new IYL(minY,i,ih);                                          ●
}                                                                       ●
```

4

Monadic Functional Reactive Programming[1]

Summary

Functional Reactive Programming (FRP) is a way to program reactive systems in functional style, eliminating many of the problems that arise from imperative techniques. In this chapter, we present an alternative FRP formulation that is based on the notion of a *reactive computation*: a monadic computation which may require the occurrence of external events to continue. A *signal computation* is a reactive computation that may also emit values. In contrast to signals in other FRP formulations, signal computations can end, leading to a monadic interface for sequencing signal phases. This interface has several advantages: routing is implicit, sequencing signal phases is easier and more intuitive than when using the switching combinators found in other FRP approaches, and dynamic lists require much less boilerplate code. In other FRP approaches, either the entire FRP expression is re-evaluated on each external stimulus, or impure techniques are used to prevent redundant re-computations. We show how Monadic FRP can be implemented straightforwardly in a *purely functional* way while preventing redundant re-computations.

4.1 Introduction

Many computer programs are *reactive*: they engage in a dialogue with their environment, responding to events as they arrive. Examples of such programs are computer games, control systems, servers, and GUI applications. Imperative techniques to create reactive systems, such as the observer pattern, lead to plethora of problems: inversion of control, non-modularity and side effects [Maier and Odersky, 2012].

[1]This chapter was published earlier as: A. van der Ploeg. Monadic Functional Reactive Programming. In *Proceedings of the '13 Symposium on Haskell*, pages 117-128, 2013.

Functional Reactive Programming (FRP) [Elliott and Hudak, 1997] is a programming paradigm to define reactive systems in functional style, eliminating many of the problems of imperative techniques. FRP has been successfully applied in many domains, such as robotics [Hudak et al., 2003; Peterson et al., 1999b,a], computer vision [Peterson et al., 2001], gaming [Courtney et al., 2003], web programming [Meyerovich et al., 2009] and graphical user interfaces [Courtney and Elliott, 2001].

The primary abstraction in FRP is a *signal* [Nilsson et al., 2002]: a value that changes over time. Traditionally, signals are modeled as mappings from points in time to values. For example, the position of the mouse can be modeled by a function that takes a number of seconds since the program started and returns the coordinates of the pointer at that time. Such signals can then be composed directly [Elliott and Hudak, 1997] or by composing *signal functions* [Courtney and Elliott, 2001], functions from signal to signal.

In this chapter, we present a novel approach to FRP called *Monadic Functional Reactive Programming* that does not model signals as mappings from points in time to values. Instead, Monadic FRP is based on the notion of a *reactive computation*: a monadic computation which may require the occurrence of external events to continue. The Monadic FRP variant of a signal is a *signal computation*: a reactive computation that may also *emit* values during the computation.

This novel formulation has two main differences with other FRP approaches:

- In contrast to signals in other FRP formulations, signal computations can *end*. This leads to a simple, monadic interface for sequencing signal phases.
- In other FRP approaches, either the entire FRP expression is re-evaluated on each external stimulus, or impure techniques are used to prevent redundant re-computations: re-computing the current value of signal while the input it depends on has not changed. Monadic FRP can be implemented straightforwardly in a *purely functional* way while preventing such redundant re-computations.

Our contributions are summarized as follows:

- A novel monadic FRP programmer interface. We demonstrate this programming model by composing a drawing program from simple components (Section 4.2).
- A comparison of the Monadic FRP programmer interface with the programmer interface of other FRP formulations (Section 4.3).
- The first purely functional FRP evaluation model which prevents redundant re-computations (Section 4.4).
- The implementation of the composition functions from the programmer interface on top of this evaluation model (Section 4.5).
- A comparison of the Monadic FRP evaluation model with other FRP evaluation models (Section 4.6).

In Section 4.7 we conclude and discuss future work. A library based on the ideas in this chapter is available as hackage package *DrClickOn*.

4.2 Programming with Monadic FRP

4.2.1 The drawing program

In this section, we demonstrate the Monadic FRP programming interface by composing a simple drawing program from small parts. The drawing program allows the user to draw boxes, change their color and delete boxes. The lifetime of each box consists of three phases:

1. *Define:* The user can define a box by holding down the left mouse button. The left-upper point of the rectangle is the mouse position when the user presses the left mouse button, the right-lower point is the mouse position when the user releases the left mouse button. While the user holds down the left mouse button, the preliminary rectangle is shown like in Figure 4.1(a).
2. *Choose color:* The user can cycle through possible colors for the box by pressing the middle mouse button, which changes the color of the box as shown in Figure 4.1(b). During this phase the box is animated so that is slowly wiggles from left to right to indicate that the color is not fixed yet. This phase ends when the user presses the right mouse button.
3. *Wait for delete:* The color and size of the box are now fixed. The user can delete the box by right double-clicking on it.

As soon as Phase 1 of a box ends, a new box can be defined. In this way there may be multiple boxes on screen, as shown in Figure 4.1(c). We develop an expression for each phase of the box, the lifetime of a box is then described by sequentially composing these phases. Finally, a combination of sequential and parallel composition is used to allow multiple boxes to be active at the same time. The entire code for this example can be obtained at `http://github.com/cwi-swat/monadic-frp`.

4.2.2 Reactive computations

The basic concept in Monadic FRP is a *reactive computation*: a monadic computation of a *single* value, which may require the occurrence of external events to continue. The type of a reactive computation is $React_g\ a$, where a is the type of the result of the reactive computation. The drawing program is created by composing the following basic reactive computations[2]:

$$\begin{aligned}&mouseDown :: React_g\ \{MouseBtn\}\\ &mouseUp\ \ \ \ \ :: React_g\ \{MouseBtn\}\\ &mouseMove :: React_g\ Point\end{aligned}$$

[2]In this chapter we use $\{a\}$ to denote $Set\ a$.

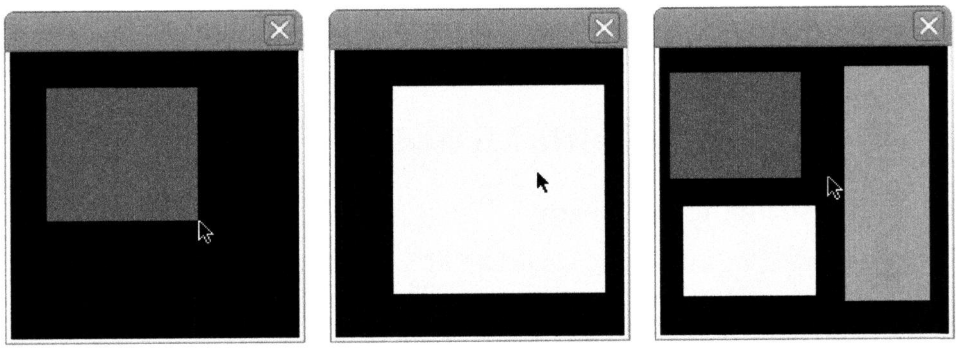


Figure 4.1: Screenshots of the simple drawing program.

```
deltaTime   :: React_g Time
sleep       :: Time → React_g ()

type Point     = (Double, Double)   -- in pixels
data MouseBtn = MLeft | MMiddle | MRight
type Time      = Double             -- in seconds
```

Here, *mouseDown* is a reactive computation that completes on the next mouse press by the user, and then returns the mouse buttons that are pressed. Typically this will be a single mouse button, but it may be that the user presses multiple buttons simultaneously, and hence the result is a *set* of buttons. Similarly, *mouseUp* returns the mouse buttons that are *released* next. The reactive computation *mouseMove* completes on the next move of the mouse, and gives the new mouse position on screen. The reactive computation *deltaTime* reports a *change in time*: the elapsed time in seconds since the last update. How fast *deltaTime* completes depends on the processing power available, as we will see later. Finally, *sleep* is the reactive computation that completes after waiting the given number of seconds. The subscript $_g$ in the type of reactive computation $React_g$ indicates the set of events that the reactive computation may deal with, and will be explained in Section 4.4.

Our drawing program is an expression where the above basic reactive computations are the leaves of the expression. The functions that are used to form this expression by converting, transforming and composing other expressions are shown in Figure 4.2. In the rest of this section, we discuss these functions and show how they are used to compose the drawing program from small components.

Reactive computations can be composed *sequentially*, yielding a new reactive computation that acts as the first reactive computation until it completes, then passes its result to a function which returns a second reactive computation, and finally acts as this second reactive computation until it completes. The function to compose reactive computations sequentially is the bind (≫=) function from the *Monad* type class. As an example, the following defines a reactive computation that decides if the user has pressed the same mouse button(s) in succession, using do notation:

```
sameClick :: React_g Bool
```

Parallel composition

first	::	$React_g\ a$	$\to React_g\ b$	$\to React_g\ (React_g\ a, React_g\ b)$
at	::	$Sig_g\ a\ y$	$\to React_g\ b$	$\to React_g\ (Maybe\ a)$
until	::	$Sig_g\ a\ l$	$\to React_g\ b$	$\to Sig_g\ a\ (Sig_g\ a\ l, React_g\ b)$
$<\!\wedge\!>$	::	$Sig_g\ (a \to b)\ l$	$\to Sig_g\ a\ r$	$\to Sig_g\ b\ (Sig_g\ (a \to b)\ l, Sig_g\ a\ r)$
indexBy	::	$Sig_g\ a\ l$	$\to Sig_g\ b\ r$	$\to Sig_g\ a\ ()$

Sequential composition

$(\gg=)$	::	$React_g\ b$	$\to (b \to React_g\ a)$	$\to React_g\ a$
$(\gg=)$	::	$Sig_g\ x\ b$	$\to (b \to Sig_g\ x\ a)$	$\to Sig_g\ x\ a$

Repetition

repeat	::	$React_g\ a$	$\to Sig_g\ a\ ()$
spawn	::	$Sig_g\ a\ x$	$\to Sig_g\ (ISig_g\ a\ x)\ ()$

Transformation

map	::	$(a \to b) \to Sig_g\ a\ r \to Sig_g\ b\ r$
scanl	::	$(a \to b \to a) \to a \to Sig_g\ b\ r \to Sig_g\ a\ r$
find	::	$(a \to Bool) \to Sig_g\ a\ r \to React_g\ (Either\ a\ r)$

Parallel element composition

dynList	::	$Sig_g\ (ISig_g\ a\ x)\ ()$	$\to Sig_g\ [a]\ ()$

Conversion

return	::	a	$\to React_g\ a$
return	::	a	$\to Sig_g\ x\ a$
done	::	$React_g\ a$	$\to Maybe\ a$
cur	::	$Sig_g\ a\ x$	$\to Maybe\ a$
emit	::	a	$\to Sig_g\ a\ ()$
always	::	a	$\to Sig_g\ a\ ()$
waitFor	::	$React_g\ a$	$\to Sig_g\ x\ a$

Figure 4.2: The types of composition, transformation and conversion functions for reactive and signal computations in Monadic FRP.

```
sameClick = do pressed  ← mouseDown
               pressed2 ← mouseDown
               return (pressed ≡ pressed2)
```

Here the function *return*, also from the *Monad* type class, converts a value into a reactive computation which immediately completes and returns the given value.

Another example of sequential composition is the following reactive computation, which completes when a given mouse button is pressed:

```
clickOn :: MouseBtn → React_g ()
clickOn b =
  do bs ← mouseDown
     if b `member` bs then return () else clickOn b
leftClick   = clickOn MLeft
middleClick = clickOn MMiddle
rightClick  = clickOn MRight
```

The basic function to compose reactive computations in *parallel* is *first*, whose type is listed in Figure 4.2. This function gives the reactive computation that runs both argument reactive computations in parallel, and completes as soon as either one of the arguments completes. The result is then the pair of the new states of both reactive computations, one of which has completed (or both when they complete simultaneously). We can use this function, for example, to create a reactive computation that given two reactive computations decides if the first completes before the second:

```
before :: React_g a → React_g b → React_g Bool
before a b = do (a', b') ← first a b
                case (done a', done b') of
                  (Just _, Nothing) → return True
                  _                 → return False
```

Where *done* is a function that given the state of a reactive computation, returns the result of this reactive computation wrapped in *Just* if the reactive computation is done, and *Nothing* otherwise.

Sequential and parallel composition can be combined to form more complex expressions. For example, the following reactive computation completes when the user has double-clicked the right mouse button, where a double-click is defined as two clicks within 200 milliseconds:

```
doubler :: React_g ()
doubler = do rightClick
             r ← rightClick `before` sleep 0.2
             if r then return () else doubler
```

4.2.3 Signal computations

The second concept in Monadic FRP is a *signal computation*, a reactive computation that may also *emit* values. A signal computation has type $Sig_g\ a\ b$, with two type arguments: the type of the values that it emits, a, and the type of the value that it returns, b. As the name suggests, the analogue to a signal computation in other FRP formulations is a signal. In contrast to a signal in other FRP formulations, a signal computation can *end*, yielding its result. Another way of looking at it is that a signal computation is a *fragment* of a signal.

To understand the usage of signal computations, consider a modal dialog in a GUI application: a pop-up window where the user must type his name before the program continues. We can model this pop-up window as a signal computation in the following way: The values that the signal computation emits are the descriptions of the appearance of the pop-up window. This description can be, for example, the current size of the pop-window and the text in the text field. When signal computation emits new descriptions, for example because the user enters letters, these descriptions should be processed and the resulting image should be drawn on screen. This signal computation completes when the user has finished entering his name, after which the pop-up disappears and the signal computation returns the name of the user.

A signal computation describes the lifetime of some object, such as a pop-up window. We call the values that a signal computation emits, such as the descriptions of the appearance of the pop-up window, the *form* of the signal computation, i.e. what can be observed from the outside. Each emission is an update to the form of the object. The *current form* is the last emitted value, and if a signal computation did not emit a value yet we say that it is *uninitialized.* When a signal computation ends, the object that it describes ends, and the result is the information to the rest of the program on how to continue, for example the name of the user. In contrast, a reactive computation cannot emit values, it just computes a value for use in the rest of the program.

The two basic functions to create a signal computation are *waitFor* and *emit*. The first, *waitFor* converts a reactive computation into a signal computation, where the resulting signal computation never emits a value (i.e. it has no form) and returns the result of the reactive computation. The second, *emit* takes a value and gives a signal computation that emits that value and then immediately returns. Like reactive computations, signal computations can be composed sequentially using $\gg\!=$, in much the same way.

As an example, consider the signal computation that models the color of the box during the Phase 2. It emits a color at the start and after each middle mouse click, until the user presses the right mouse button, after which it returns the number of colors it emitted. This signal computation is defined as as follows:

```
cycleColor :: Sig_g Color Int
cycleColor = cc colors 1 where
  cc (h : t) i = do
    emit h
    r ← waitFor (middleClick `before` rightClick)
    if r then cc t (i + 1) else return i
```

Where *colors* is an infinite list of colors (not shown).

Another way to create a signal computation is to *repeat* a reactive computation. The function to do this is unsurprisingly named *repeat*, and gives the signal computation that indefinitely repeats the given reactive computation, each time emitting the resulting value. This signal computation never ends, and hence its result, (), will never be reached. An example is the signal computation that emits the current mouse positions:

```
mousePos :: Sig_g Point ()
mousePos = repeat mouseMove
```

Signal computations can be transformed by functions such as *map*, *scanl* and *find* that are familiar from list programming. As an example, the following signal computation emits the preliminary rectangles in Phase 1 of a box, given the left-upper point of the rectangle.

```
curRect :: Point → Sig_g Rect ()
curRect p1 = map (Rect p1) mousePos
data Rect = Rect {leftup :: Point, rightdown :: Point}
```

The list function *scanl* is similar to *foldl*, but it returns a list of successive reduced values instead of a single value. The signal transformation function *scanl* works analogously, it emits a new reduced value each time the given signal emits. Using *scanl*, we define a signal that on each update, emits the number of seconds since it started:

```
elapsed :: Sig_g Time ()
elapsed = scanl (+) 0 (repeat deltaTime)
```

Using *elapsed*, we implement *animation* by transforming each point in time to the frame of the animation at that time. As an example, the following signal emits the rectangle animation in Phase 2:

```
wiggleRect :: Rect → Sig_g Rect ()
wiggleRect (Rect lu rd) = map rectAtTime elapsed
  where rectAtTime t = Rect (lu +. dx) (rd +. dx)
                     where dx = (sin (t * 5) * 15, 0)
```

Where +. (not shown) is the vector addition operator for points.

The last list-like function that we use in our example, *find*, gives a reactive computation that completes as soon as the given signal computation emits a value on which the given predicate holds. As an example, the following function gives a reactive computation which completes as soon as the argument signal computation emits a point inside a given rectangle:

```
posInside   :: Rect → Sig_g Point y
               → React_g (Either Point y)
posInside r = find (‘inside‘r)
inside :: Point → Rect → Bool
```

Signal computations and reactive computations can be composed in parallel by two functions: *at* and *until*. The first, *at*, takes a signal computation and a reactive computation, and returns the current form of the signal computation at the time the reactive computation completes. For example, the mouse position at the next left mouse click is defined as follows:

```
firstPoint :: React_g (Maybe Point)
firstPoint = mousePos ‘at‘ leftClick
```

The second, *until*, takes a signal computation and a reactive computation, and runs the signal computation until the reactive computation completes. Like *first*, the result of *l* ‘*until*‘ *a* is the pair of the new state of *l* and the new state of *a*. For example, the following gives the preliminary rectangles in Phase 1 until the user releases the left mouse button.

```
completeRect :: Point → Sig_g Rect (Maybe Rect)
completeRect p1 = do (r, _) ← curRect p1 ‘until‘ leftUp
                     return (cur r)
```

Where *leftUp* (not shown) is defined analogously to *leftDown*. The function *cur* gives the current form of a signal computation, i.e. the last value it emitted.

By composing *firstPoint* and *completeRect* sequentially, we define the signal computation that emits the rectangles in Phase 1:

```
defineRect :: Sig_g Rect Rect
defineRect = do Just p1 ← waitFor firstPoint
                Just r  ← completeRect p1
                return r
```

The function to compose two signal computations in parallel is <^>, which takes a signal computation emitting functions and a signal computation emitting values, and gives the signal computation that emits the results obtained by feeding the values to the functions over time. More precisely, the signal computation *f* <^> *x* operates as follows:

- Wait until both input signals have started emitting values.
- On each emission from either the function signal computation or the value signal computation we apply the latest value to the latest function and emit the resulting value.
- Repeat the previous step until either of the signals end.

The result of the signal computation f <^> x is the new state of both input signal computations, one of which has ended.

We can use this operator to compose the signal computation of the rectangle and the signal computation of the color in parallel, to obtain a signal computation which describes Phase 2 of a box:

```
chooseBoxColor :: Rect → Sig_g Box ()
chooseBoxColor r =
  do always Box <^> wiggleRect r <^> cycleColor
     return ()
data Box = Box Rect Color
```

The operator <^> binds less strongly than function application. The function *always* takes a value and gives a signal computation that emits that value and then never emits again and never ends. In this way, the current form of *always* x is always x. The signal computation *chooseBoxColor* r ends when the user presses the right mouse button, as this causes *cycleColor* to end, which in turns ends the compositions using <^>.

The functions <^> and *always* are inspired by the *Applicative functor* type class [Mcbride and Paterson, 2008]: the function <^> corresponds to <*> and *always* corresponds to *pure*. The difference is that the *Applicative* type class operates on the last argument of a type constructor, but here we want <^> to operate on the emitted arguments, i.e. the first type argument of the type constructor Sig_g. In this way Monads are used for sequential composition, and an Applicative functor-like interface is used for parallel composition.

Another interesting way to compose signal computations in parallel it to use one as a *time index* for the other. This means that we sample the form of the first signal computation each time the second signal computation emits. For instance, *mousePos* ‘*indexBy*‘ *repeat doubler* is the signal that emits the mouse positions at the times when the user right-double clicks. We can use this operator to define a reactive computation that completes as soon as the user double right clicks on a given rectangle:

```
drClickOn :: Rect → React_g (Maybe Point)
drClickOn r =
  posInside r (mousePos `indexBy` repeat doubler)
```

We now have all the ingredients to define the behavior of a single box, as we have defined each phase of the box, so we only have to compose them sequentially:

```
box :: Sig_g Box ()
box = do r ← map setColor defineRect
         chooseBoxColor r
         waitFor (drClickOn r)
         return ()
  where setColor r = Box r (head colors)
```

This signal computation describes the entire lifetime of a box, its form is appearance of the box and the signal computation ends when the user deletes the box.

4.2.4 Dynamic lists

We now have the signal computation for a single box, but we would like our drawing program to allow the user to draw *multiple* boxes. Luckily, signal computations are just *values*, and hence like reactive computations, they can be *repeated*. For this we introduce the function *spawn* which takes a signal computation and returns a signal computation that emits *initialized signals*: signal computations which are initialized, i.e. the first form of the object it describes is known. In this way, we can define a signal that emits initialized signals of the boxes that the user creates as follows:

$$\begin{aligned}&\mathit{newBoxes} :: \mathit{Sig}_g\ (\mathit{ISig}_g\ \mathit{Box}\ ())\ ()\\&\mathit{newBoxes} = \mathit{spawn}\ \mathit{box}\end{aligned}$$

This signal computation starts a *box* computation, and as soon as it emits its first value, *newBoxes* emits the initialized signal corresponding to that box. Afterwards, a new box computation is started and the process repeats.

These initialized signals can then be composed *parallel*, so that there are multiple boxes on the screen, and the user can interact with all of them. For this we introduce the function *dynList*, which takes a signal computation emitting initialized signals, and composes these initialized signals in parallel. The result is a *dynamic list*: a list that changes over time. The signal computation that describes this dynamic list emits the lists of boxes, namely the current forms of all boxes that are active at that time. When a new box is defined it is added to the list and when a box is deleted, i.e. its initialized signal ends, it is removed from the list. In this way, we can define the top-level expression of our drawing program simply as:

$$\begin{aligned}&\mathit{boxes} :: \mathit{Sig}_g\ [\mathit{Box}]\ ()\\&\mathit{boxes} = \mathit{dynList}\ \mathit{newBoxes}\end{aligned}$$

4.2.5 Time-branching

Monadic FRP has *time-branching semantics*: we can observe the values a signal computation emits when given some event occurrences, and afterwards we can still observe what values the orignal signal computation emits when given other event occurrences. These time-branching semantics are also known as *shallow causality* [Jeffrey, 2013]. They are also supported by Arrowized FRP [Courtney and Elliott, 2001], by "freezing" signal transformers.

We can use these time-branching semantics, for example, to easily implement multiple tabs in our drawing program. The user can then duplicate its current drawing into two tabs, modify the drawing and switch back to the tab holding the original drawing, which can then again be modified. Each of these tabs is described by a signal computation, but only one observes the current event occurrences. Duplication of a tab is then simply duplicating the signal computation in the list of tabs, and switching between tabs controls which tab observes the current event occurrences and is rendered to the screen. The code for this tabbed drawing program is not included in this chapter for space reasons, but can be seen online. As we show in Section 4.6, time-branching semantics are only supported by purely functional evaluation mechanisms.

4.3 Comparison with other FRP programmer interfaces

In this section, we compare the Monadic FRP programmer interface to other FRP programmer interfaces. We compare mainly with Arrowized FRP [Courtney and Elliott, 2001], more precisely the Yampa [Nilsson et al., 2002] framework, and discuss other FRP formulations in passing.

In Arrowized FRP, signals are not first class entities: they cannot be created or manipulated directly. Instead, the basic concept in Arrowized FRP is a *signal function*: a mapping from input signal to output signal. A signal function has type $SF\ a\ b$, where a is the type of the input signal and b is the type of the output signal. Signal functions can then be composed using the *Arrow* type-class [Hughes, 2000]. We assume basic familiarity with this type-class and its notation [Paterson, 2001] in the rest of this section. It should be noted that the examples in this section are cherry-picked to show the advantages of Monadic FRP and hence may give a skewed impression.

In contrast to signal computations in Monadic FRP, signals in Arrowized FRP cannot end. Another difference is that signals in Arrowized FRP must emit a value for *each* input value. For this reason, among others, Arrowized FRP has the concept of an *event source*: a signal that emits values of the option type $Event\ a$. An event source emits *NoEvent* when there is no event, and an $Event\ a$, where a is the information associated with the event, when there is an event.

Figure 4.3 shows the implementation of the *cycleColor* signal (function) in both Monadic and Arrowized FRP. In the Arrowized version, *cycleColor* is a signal function which takes a signal producing mouse press events, and transforms it into a signal producing a color and an event of type *Int*. This event occurs when the user is done choosing colors, and then contains the number of different colors the user considered. Notice that when such an event occurs, the signal does not stop as in the Monadic FRP formulation of *cycleColor*, because signals cannot end.

4.3.1 Advantages of Monadic FRP

Implicit routing

The most obvious difference when considering the code in Figure 4.3 is the difference between do notation and arrow notation. To compose signal functions in arrow notation, the programmer needs to route the output of component arrows and the input signal into the input of component arrows and the output signal. In other FRP

```
cycleColor :: Sig_g Color Int
cycleColor = cc colors 1 where
  cc (h : t) i = do
    emit h
    r ← waitFor (middleClick ‘before‘ rightClick)
    if r then cc t (i + 1) else return i
```

(a) Monadic FRP

```
cycleColor :: SF MouseDown (Color, Event Int)
cycleColor = cc colors 1 where
  cc (h : t) i = switch (proc md → do
    mc ← notYet ⋘ middleClick −≺ md
    rc  ← rightClick           −≺ md
    returnA −≺ ((h, tag rc i), mc)
    )(λ_ → cc t (i + 1))
```

(b) Arrowized FRP.

Figure 4.3: Side-by-side comparison of *cycleColor* in Monadic and Arrowized FRP.

formulations, such as Classic FRP [Elliott and Hudak, 1997], such wiring is also necessary, but by composing functions instead of arrows. In Monadic FRP, this routing is *implicit*, reducing boilerplate code and visual clutter.

Easier sequential composition

Because signals in Arrowized FRP cannot end, a different approach is taken to describe signals which consist of multiple phases. For this a variety of *switching combinators* is used, which allow us to switch from one signal function to another, when a certain event occurs. The most basic switching combinator in Yampa is *switch*, which has the following type:

$$switch :: SF\ a\ (b, Event\ c) \rightarrow (c \rightarrow SF\ a\ b) \rightarrow SF\ a\ b$$

The first argument to this combinator is a signal function transforming a signal of type a into a signal giving a combination of something of type b and an event of type c. The second argument is a continuation function: given a value of type c it will produce a new signal function. The result of the *switch* combinator is a signal function from a to b, which first behaves as the first argument signal function, except that the *Event* c is not visible from the outside. When this first argument signal function generates an event of type c, the continuation function is called. Afterwards, the resulting signal function is *switched* to: the result of the *switch* combinator will behave as this signal function.

In our example in Figure 4.3(b), the signal function *cycleColor* is intended to be switched out when a right mouse event occurs. However, a right mouse click event does not contain the color count. For this reason, we have to set the associated data of the mouse press event to the color count, by means of the $tag :: Event\ a \to b \to Event\ b$ combinator. In Monadic FRP, such explicit transformation of the associated data of events is not necessary.

All Yampa switching combinators come in two flavors:

- *immediate*, in which case the output at the time of switching is determined by the signal function being switched to.
- *decoupled*, in which case the output at the time of switching is determined by the original signal function.

In Monadic FRP, signals can *either* end or emit a value, but not both at the same time. Hence, the distinction between immediate and decoupled switching is not present in Monadic FRP, and the associated subtleties disappear.

In our example, the *notYet* combinator is used to delay the switching event by one time-step. Without an invocation of *notYet*, the program will go into an infinite loop when the middle mouse button is pressed. The reason for this is that the new signal function is again an application of *cc*, which will then immediately switch again, since the input signal currently indicates that the middle mouse button is down. In Monadic FRP, such event suppression is not necessary, because an event can only be consumed once by a reactive computation. For example, $rightClick \gg rightClick$ will complete after two right clicks, not one.

Another benefit of Monadic FRP is that signal computations decide themselves that they end, whereas with *switching* combinators this is decided by the context. Hence, in Arrowized FRP, if a programmer intends a signal function to be switched out after a certain event occurs, the programmer must still provide the signal function after this event. In Monadic FRP this is not necessary: the programmer can force the context to "switch".

A simpler way of creating dynamic lists

In Yampa, creating dynamic lists requires the following *parallel switching combinator*[3]:

$$\begin{aligned} pSwitchList ::\ & [SF\ a\ b] \to SF\ (a, [b])\ (Event\ c) \\ & \to ([SF\ a\ b] \to c \to SF\ a\ [b]) \to SF\ a\ [b] \end{aligned}$$

This switching combinator requires three arguments:

- The initial list of signal functions.
- A signal function that transforms the input and the current list of values to a switching event.

[3]This switching combinator is the list version of *pSwitch*. We have chosen to use this specialized combinator in the comparison, since *dynList* also deals with lists.

```
boxes :: Sig_g [Box] ()
boxes = dynList (spawn box)
```

(a) Monadic FRP.

```
type BoxSF = SF GUIIn (Box, Event ())
boxes :: SF GUIIn [Box]
boxes = boxes' [] ⋙ arr (map fst) where
  boxes' i = pSwitchList i
     (newBox *** arr toEv ⋙ arr choose ⋙ notYet)
     (λe l → boxes' (mutateList e l))
choose (a, b) = merge (fmap Left a) (fmap Right b)
toEv l = let l' = map (isNoEvent ∘ snd) l
         in if and l' then NoEvent else Event l'
mutate :: [BoxSF] → Either (BoxSF) [Bool] → [BoxSF]
mutate l (Left b) = b : l
mutate l (Right l') = map fst (filter snd (zip l l'))
box    :: BoxSF
newBox :: SF GUIIn (Event BoxSF)
```

(b) Arrowized FRP.

Figure 4.4: Dynamic list in the drawing program.

- A continuation function that when given the current list of signal functions and the value associated with a switching event, returns the new signal function.

The code for the dynamic list of boxes in Monadic and Arrowized FRP is shown in Figure 4.4. In the Arrowized FRP code, we assume that the signal function for a single box produces the current form of the box and an event indicating that the box has ended. The difficulty in creating the dynamic list then lies in wiring the switching events of all boxes and the switching event for creating a new box together, and then picking the resulting switching event information apart again in the continuation function. In Monadic FRP, such wiring is not necessary.

4.3.2 Disadvantages of Monadic FRP

While Monadic FRP has several advantages over other FRP formulations, it also has some disadvantages. In particular, to share the computation of a signal which occurs more than once in an expression, we have to resort to a manual invocation of a memoization function. This is not necessary in several other FRP formulations, including Arrowized FRP.

A related disadvantage is that it is unclear how to declare mutually dependent signals in Monadic FRP, such as two sliders in a temperature conversion application that influence each other. In Arrowized FRP, such mutually dependent signals can simply be declared by recursive arrow notation.

4.4 Evaluating Monadic FRP expressions

In this section, we show how reactive and signal computations are evaluated in a simple, straightforward manner.

4.4.1 Event requests and occurrences

Central to Monadic FRP evaluation is the notion of an event: a stimulus from the environment. Reactive computations *request* the observation of such events, an interpreter then observes such events and passes the event *occurrence* back. We model event requests and occurrences with the following data type:

```
data Event a = Request | Occurred a
```

Where the argument to the constructor *Occurred* is the associated data of the occurred event. For simplicity, event requests and occurrences are defined using the same data type in our approach.

To make things more concrete, the following events are used in the program drawing example:

```
data GUIEv = MouseDown    (Event {MouseBtn})
           | MouseUp      (Event {MouseBtn})
           | MouseMove    (Event Point)
           | DeltaTime    (Event Time)
           | TryWait Time (Event Time)
           deriving (Eq, Show, Ord)
```

When a reactive computation, for example, wants to know the next mouse button that is pressed, it passes the event request *MouseDown Request* to the interpreter of the reactive expression. This interpreter, from now on called the *reactive interpreter*, then waits for the next mouse press and returns the event occurrence, for example *MouseDown (Occurred {MLeft, MMiddle})*, which indicates that the user pressed the left and middle mouse buttons simultaneously.

The reactive interpreter can wait for multiple events in parallel, and hence we pass a *set* of event requests to it. As soon as at least one of these events occurred, the reactive interpreter responds by returning the occurred event(s). This response is a set of event occurrences, since multiple events may occur simultaneously. Since event requests and occurrences are modeled by the same datatype, we use the following type aliases to make the distinction clear:

```
type EvReqs e = {e}  -- event requests
type EvOccs e = {e}  -- event occurrences
```

4.4.2 Reactive computations

Using this basic terminology introduced above, a state of a reactive computation is defined as follows:

```
data React e a
  = Done a
  | Await (EvReqs e) (EvOccs e → React e a)
```

If a reactive computation is done, it is in state *Done* and carries the resulting value of the computation of type a. Otherwise, it *awaits* at least one event occurrence from its set of event requests. As soon as one of these events occur, or multiple events occur simultaneously, the event occurrences can be passed to the *continuation function.* This continuation function then processes the event occurrences and returns the new state of the reactive computation. The type e is the type of the events that the reactive computation may request and process. In our drawing program in Section 4.2, the type of events is *GUIEv*, hence the type $React_g$ that is used throughout Section 4.2 is defined as follows:

```
type React_g = React GUIEv
```

The basic reactive computations *mouseDown*, *mouseUp*, *mouseMove*, *deltaTime* and *tryWait* are then defined as follows:

```
mouseDown = req (MouseDown Request) >>= get
  where get (MouseDown (Occurred s)) = return s
 ⋮
tryWait t = req (TryWait t Request) >>= get
  where get (TryWait _ (Occurred t)) = return t
req :: e → React e e
req a = Await (singleton a) (Done ∘ head ∘ elems)
```

Here, *req* is a function that given an event request gives the reactive computation that returns the next event occurrence that satisfies this request. The function *elems* converts a set to a list. Notice that the continuation function of a reactive computation is called with the set of event occurrences *which it awaits.* If there are no event occurrences which the reactive computation awaits, then the continuation function will *not* be called. Since *mouseDown* awaits only *MouseDown* events, we can be sure that the pattern match *MouseDown* (*Occured* s) cannot fail. The same reasoning holds for the patterns in the other basic reactive computations.

4.4.3 Evaluating reactive computations

In essence, our evaluation model is a purely functional way to use *blocking-IO multiplexing*: the program is organized as a main loop that first decides which events should be listened for, then waits for at least one of these events to occur, and finally processes the event(s) that occurred. Waiting for several events in parallel can be done by means of for example the Unix `select` or Linux `epoll` method[4], which take a set of file-descriptors and waits for one of them to become ready for reading or writing. Another example is the `waitEvent` method of the Simple Directmedia Layer[5], which waits for a user input event, such as a mouse-click or keystroke. The main loop in our approach is the reactive interpreter which interprets the top-level reactive or signal computation.

The interpreter for reactive computations is defined as follows:

$$\begin{array}{l}
\mathit{interpret} :: \mathit{Monad}\ m \Rightarrow (\mathit{EvReqs}\ e \rightarrow m\ (\mathit{EvOccs}\ e)) \\
\qquad\qquad \rightarrow \mathit{React}\ e\ a \rightarrow m\ a \\
\mathit{interpret}\ p\ (\mathit{Done}\ a) \quad\ \ = \mathit{return}\ a \\
\mathit{interpret}\ p\ (\mathit{Await}\ r\ c) = p\ r \gg\!= \mathit{interpret}\ p \circ c
\end{array}$$

Here p is a function that takes a set of event requests and waits for any of these events to occur in the monad m, which is for example the *IO* monad. The drawing program described in Section 4.2 can be run in an interpreter which uses the `waitEvent` method of the Simple Directmedia Layer to define the function p. After an interesting event occurred, p returns the set of event occurrences, which is then fed back into the reactive computation. This process continues until the reactive computation completes and returns a value.

The reactive computation that is interpreted consists solely of the sequential and parallel composition of basic reactive computations, other composition operators are defined in terms of these two composition operators. As an example, consider the following reactive expression:

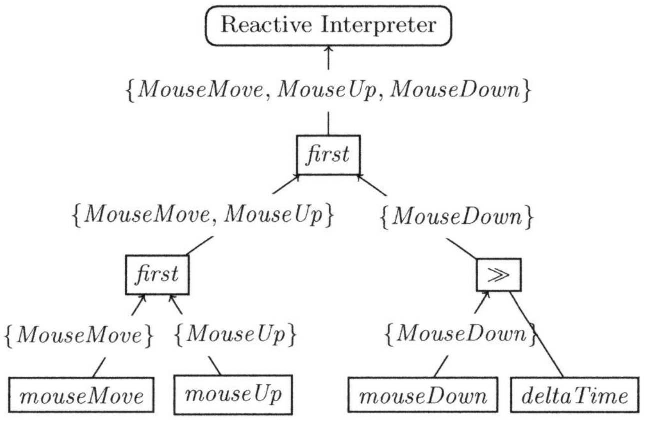
$$\begin{array}{l}
\mathit{first}\ (\mathit{first}\ \mathit{mouseMove}\ \mathit{mouseUp}) \\
\qquad (\mathit{mouseDown} \gg \mathit{deltaTime})
\end{array}$$

Figure 4.5(a) shows the tree of this expression and which event requests are propagated *upwards* to the reactive interpreter. When composing reactive computations sequentially, using $\gg\!=$, the event requests of the composed expression are just the event requests of the first argument. Hence, the event requests of $\mathit{mouseDown} \gg \mathit{deltaTime}$ are just $\{\mathit{MouseDown}\}$. When composing reactive computations in parallel, using *first*, the event requests of the composed expression are the union of the event requests of both arguments. In this way the reactive interpreter knows exactly which events to wait for.

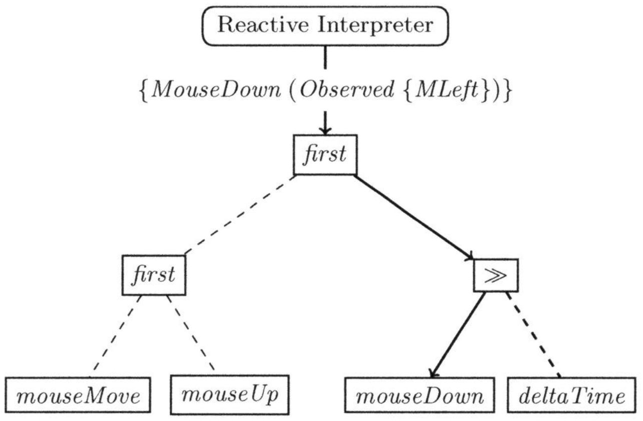
The reactive interpreter then waits for events from such a set of event requests. When one event occurred, or multiple events occurred simultaneously, the set of

[4]For more information see `man select` or `man epoll`.

[5]`http://libsdl.org`



(a) How event requests are propagated upwards.



(b) How an event occurance is propagated downwards.

Figure 4.5: The tree of the expression
first (*first mouseMove mouseUp*) (*mouseDown* $\gg$ *deltaTime*).

event occurrences is passed to the continuation function of the reactive computation. If the reactive computation is a sequential composition, then the event occurrences are simply passed to the first argument. When the reactive computation is a parallel composition, the set of event occurrences is passed to the argument(s) that await any of these events.

Figure 4.5(b) shows how an event occurrence, stating that the left mouse button was pressed, is propagated *downwards.* Notice that the entire left leg of the tree is not updated in this process, since it did not await this particular event. In this way, the evaluation avoids unnecessary re-computations, by updating only those components that await the occurred events.

After processing this event occurrence, the reactive computation proceeds as the reactive computation:

$$\mathit{first}\ (\mathit{first}\ \mathit{mouseMove}\ \mathit{mouseUp})\ \mathit{deltaTime}$$

Hence, the reactive expression is *dynamic*: each sub-expression may change after each update. This new expression leads to different event requests than the original expression, namely the set $\{\mathit{MouseMove}, \mathit{MouseUp}, \mathit{DeltaTime}\}$. In this way the events in which the reactive computation is interested in can also change over time.

4.4.4 Time semantics

In our set of GUI events, there are two events that deal with time: *DeltaTime* and *TryWait*. The first, *DeltaTime*, asks to observe *any* change in time and returns the change in time since the previous update of the reactive interpreter. The second, *TryWait*, works similarly, but takes an argument that indicates the time it wants to wait. The result of such a *TryWait* request is also the change in time since the last update of the reactive interpreter. The difference between the two lies in how they are handled: *DeltaTime* tells the reactive interpreter to respond as quickly as possible, whereas *TryWait* tells the reactive interpreter to try and wait the given time before responding. Hence, when only *TryWait* requests are given to the reactive interpreter, then the reactive interpreter just waits for time to pass, without wasting CPU cycles needlessly updating the reactive expression.

An event request *TryWait* asks the reactive interpreter to wait for the given time, but there may be another event request that can be answered earlier. In that case, the interpreter cannot wait the given time and must respond. Hence, the time it takes for the event *TryWait* to occur might be less than the requested amount of time.

As an example usage of *TryWait*, consider the *sleep* reactive computation, which completes after the given number of seconds:

```
sleep t = do t′ ← tryWait t
             if t′ ≡ t then return () else sleep (t − t′)
```

Notice that testing for equality here is safe, because the result of *TryWait* request may be less than the requested time, but not more. Hence, we can be sure that *sleep* 1.1

never completes earlier or simultaneously to *sleep* 1. In other purely functional implementations of FRP, such exact timing is not available: testing for equality on time is unsafe, since the precision of timing depends on how often the signal is sampled.

Such exact timing is achieved by handling the event requests in the reactive interpreter as follows:

- Compute the maximum time to wait, which is the minimum of the times given to *TryWait* event requests. It is infinity if there are no *TryWait* requests.

- See if the maximum time to wait, t, is smaller than the time since the last update, t'. If so, we construct only *TryWait* and *DeltaTime* occurrences with t as their associated data and return them, the other steps will not be executed on this iteration. The next update will have time difference $t' - t$ plus the new time difference. In this way, the result of a *TryWait* request will never be more than the requested time.

- Otherwise, wait for an event from the set of event requests, using the maximum time to wait as a *timeout* duration. Blocking I/O multiplexing functions such as `select` and SDL's `waitEvent` usually allow such a timeout duration. If there is a *DeltaTime* request, then 0 is passed as the time duration, and the events that are currently available will be returned, i.e. the blocking I/O multiplexing function will not block.

- Construct and return the set of event occurrences, including the occurrences of *TryWait* and *DeltaTime*, which get the time since the last update of the reactive interpreter.

Thanks to these semantics, the drawing program will simply wait for the next mouse click or mouse move when there are no animated boxes currently on screen. If one of the boxes is animated (in Phase 2), then the reactive interpreter updates the animation as quickly as possible so that the animation is as smooth as possible. In this way, the animation is *conceptually continuous*: we describe it as if the animation is continuous, abstracting from how often the animation is actually sampled.

4.4.5 Evaluating signal computations

Evaluation of a signal computation is very much the same as evaluation of a reactive computation, since signal computations are defined in terms of reactive computations as follows:

```
newtype Sig  e a b = Sig (React e (ISig e a b))
data    ISig e a b = a :| Sig e a b
                   | End b
```

Here the type *Sig* is the type of a signal computation and *ISig* is the type of an initialized signal, i.e. a signal computation of which the first form is known or which has already ended. Signal computations and initialized signals are defined mutually recursively, a signal computation is a reactive computation of the initialized signal, and the tail of an initialized signal is again a signal computation. The argument e is the type of events that can be handled inside the signal computation, a is the type of the values that it emits and b is the type of its result. The signal computation and initialized signals in Section 4.2 are specialized to *GUIEv*, i.e.:

```
type Sig_g  = Sig GUIEv
type ISig_g = ISig GUIEv
```

The interpreter for signal computations uses the interpreter for reactive computations to evaluate a signal computation to its corresponding initialized signal. Additionally, the values that are emitted by the signal computation are processed. For example, the interpreter of our example in Section 4.2 draws each emitted list of boxes on screen. The signal computation interpreter is defined as follows:

```
interpretSig :: Monad m ⇒ (EvReqs e → m (EvOccs e))
                 → (a → m ()) → Sig e a b → m b
interpretSig p d = interpretSig' where
  interpretSig' (Sig s)    = interpret p s ≫= interpretISig
  interpretISig (h :| t)   = d h ≫ interpretSig' t
  interpretISig (End a)    = return a
```

Here the new argument d is the function which processes each new emission of the signal computation.

4.4.6 Sharing computation results

If a reactive or signal computation occurs multiple times in an expression, then standard evaluation techniques may lead to a source of inefficiency. The simplest example of this is:

```
first x x
```

When an event occurs that x is interested in, then the evaluation of x to its new state will be performed twice. To solve this problem we introduce a *memoization* function, as is also done in other FRP approaches [Elliott, 1998]:

```
memo :: Ord e ⇒ React e a → React e a
```

In this way, we can rewrite our example to eliminate the potential problem:

```
let x' = memo x in first x' x'
```

We also introduce a memoization function for signal computations, that applies memoization on the reactive computation of the initialized signal *and* on the signal computation that is the tail of that initialized list (if any).

```
memoSig :: Ord e ⇒ Sig e a b → Sig e a b
```

The need for invocations of memoization functions is not necessary in some other FRP approaches, such as Arrowized FRP. Hence, this is a disadvantage of Monadic FRP.

4.5 Implementing composition functions

In this section, we show how a selection of the composition functions from Figure 4.2 are implemented. In this way, this section shows the semantics of the programming model explained in Section 4.2 by building on the basic evaluation mechanism explained in Section 4.4. The definition of the composition operators is mostly straightforward: the entire Monadic FRP library consists of just 137 lines of code, excluding blank lines (not including the drawing program which consists of 108 lines of code and the interface to SDL which consists of 109 lines of code). The structure of this section reflects the structure of Section 4.2. We first show the implementation of sequential and parallel composition of *reactive* computations. Afterwards, we show how these can be used to implement composition functions for signal computations, and finally we show how dynamic lists are implemented.

4.5.1 Basic composition operators

The basic composition operators in Monadic FRP are the *sequential* and *parallel* composition of *reactive computations*, all other composition and transformation operators are defined using these two basic composition operators.

Sequential composition of reactive computations

Sequential composition of reactive computations is defined as an instance of the Monad type class:

```
instance Monad (React e) where
  return          = Done
  (Await e c) >>= f = Await e (λx → c x >>= f)
  (Done v)    >>= f = f v
```

If the first reactive computation awaits some event, then its next state is again sequentially composed with f. This process repeats until the first reactive computation completes, after which the function f will be called with the result of the reactive computation, and the new reactive computation will be executed.

Parallel composition of reactive computations

Recall that parallel composition of reactive computations is achieved using *first*, which runs two reactive computations in parallel until either completes, and then gives the new state of both reactive computations. Its definition is as follows:

```
first l r = case (l, r) of
  (Await el _, Await er _) →
     let e   = el `union` er
         c b = first (update l b) (update r b)
     in Await e c
  _ → Done (l, r)
```

If both reactive computations await some event, then *first* waits for the union of their event requests, as shown in Figure 4.5(a). Then, on an event occurrence, *first* updates both reactive computations to their next state and calls *first* again, which then checks again if both reactive computations await some event. If this is not the case, then at least one of the reactive computations must have completed, and the state of both reactive computations is returned.

As shown in Figure 4.5(b), only those reactive computations that await an event that occurred should be updated. This is done by the function *update* that is used in the above definition of *first*. This function returns the new state of a reactive computation given a set of event occurrences. If the reactive computation awaits some of the events that occurred, then *update* obtains the new state of a reactive computation by calling its continuation function. Otherwise, the new state is simply the old state. The definition of *update* is as follows:

```
update :: Ord e ⇒ React e a → EvOccs e → React e a
update (Await r c) oc | oc′ ≢ empty = c oc′
  where oc′ = oc ‘filterOccs‘ r
update r _ = r
```

Here, *filterOccs*(not shown) filters the event occurrences that the reactive computation awaits from the set of event occurrences. If the resulting set of event occurrences is empty, then the reactive computation did not await in any of the events that occurred.

4.5.2 Signal computation composition functions

Since signal computations are defined in terms of reactive computations, the composition functions dealing with signal computations are implemented by combining the sequential and parallel composition of reactive computations in various ways. Figures 4.6 and 4.7 shows the definition of a selection of these signal computation composition functions and Figure 4.8 shows the definition of the conversion functions.

Signal computations and initialized signals are mutually recursively defined data types, so functions dealing with signal computations often alternate between processing a signal computation and processing an initialized signal. In the code this can be seen, for example, in the function *map*, which obtains the initialized signal and then calls *imap*, which is like *map*, but on initialized signals. The function *imap* processes the initialized signal, and calls *map* again to process the tail, which is a signal computation. The same pattern arises in the sequential composition of signal computations, and in the functions *scanl*, *until* and *res*.

The signal computation *l‘until‘ a*, splits the signal computation *l* in two: *l‘until‘ a* is the part of the signal computation before *a* completes, and the result of *l‘until‘ a* is the signal computation after *a* completed. If the signal computation *l* was initialized before *a* occurred, i.e. it had already emitted its first value, then the signal computation after *a* should *not* be an uninitialized signal computation. For instance, the result of *mousePos‘until‘ leftClick*, the mouse position *after* the left click, should not be uninitialized, but should start with the emission of the last mouse position before the left click. Hence, the result of *until* differs depending on if the signal computation

Sequential composition

```
instance Monad (Sig e a) where
  return = emitAll ∘ End
  (Sig l) >>= f = Sig (l >>= ib)
    where ib (h :| t)   = return (h :| (t >>= f))
          ib (End a)  = let Sig x = f a in x
instance Monad (ISig e a) where
  return = End
  (End a) >>= f = f a
  (h :| t)   >>= f = h :| (t >>= emitAll ∘ f)
```

Repetition

```
repeat :: React e a → Sig e a ()
repeat x = xs where xs = Sig (liftM (:| xs) x)
spawn :: Sig e a r → Sig e (ISig e a r) ()
spawn (Sig l) = repeat l
```

Transformation

```
map :: (a → b) → Sig e a r → Sig e b r
map f (Sig l)    = Sig (liftM (imap f) l)
imap f (h :| t)  = f h :| map f t
imap f (End a) = End a
scanl :: (a → b → a) → a → Sig e b r → Sig e a r
scanl f i l          = emitAll (iscanl f i l)
iscanl f i (Sig l) = i :| (waitFor l >>= lsl)
  where lsl (h :| t)   = scanl f (f i h) t
        lsl (End a) = return a
```

Figure 4.6: Implementation of sequential composition, repetition, transformation functions.

Parallel composition

```
until :: Ord e ⇒    Sig e a r → React e b →
           Sig e a (Sig e a r,    React e b)
until (Sig l) a = waitFor (first l a) ≫= un where
   un (Done l, a) = do (l, a) ← emitAll (l ‘iuntil‘ a)
                       return (emitAll l, a)
   un (l, Done a) = return (Sig l, Done a)
iuntil :: Ord e ⇒ ISig e a r →        React e b →
          ISig e a (ISig e a r,       React e b)
iuntil (End l)     a = End (End l, a)
iuntil (h :| Sig t) a = h :| Sig (liftM cont (first t a))
   where cont (Done l, a) = l ‘iuntil‘ a
         cont (t, Done a) = End (h :| Sig t, Done a)
(<^>) :: Ord e ⇒ Sig e (a → b) l → Sig e a r →
         Sig e b ( Sig e (a → b) l,   Sig e a r)
l <^> r = do (l, r) ← waitFor (bothStart l r)
             emitAll (imap (uncurry ($)) (pairs l r))
bothStart ::    Ord e ⇒  Sig   e a l → Sig   e b r →
                React e (  ISig e a l,    ISig e b r)
bothStart l (Sig r) =   do (Sig l, r) ← res (      l ‘until‘ r)
                           (Sig r, l) ← res (Sig r ‘until‘ l)
                           return (done′ l, done′ r)
pairs :: Ord e ⇒          ISig e a l →     ISig e b r →
         ISig e (a, b) (ISig e a l,        ISig e b r)
pairs (End a)    b         = End (End a, b)
pairs a            (End b) = End (a, End b)
pairs (hl :| Sig tl) (hr :| Sig tr) = (hl, hr) :| tail
   where tail = Sig (liftM cont (first tl tr))
         cont (tl, tr) = pairs (lup hl tl) (lup hr tr)
         lup _ (Done l) = l; lup h t = h :| Sig t
```

Figure 4.7: Implementation of and parallel composition functions.

```
emitAll   = Sig ∘ Done; emit a = emitAll (a :| return ())
always a = emit a ≫ hold; waitFor a = Sig (liftM End a)
hold       = waitFor never where never = Await empty ⊥
res (Sig l)  = l ≫= ires
ires (_ :| t) = res t; ires (End a) = Done a
done (Done a)                = Just a; done _ = Nothing
cur (Sig (Done (h :| _))) = Just h; cur _   = Nothing
done′ = fromJust ∘ done
```

Figure 4.8: Implementation of conversion functions.

was initialized before the reactive completes. If this is this case, then *iuntil* ensures that the signal computation after the reactive computation completes starts with the last emission before the reactive computing completed.

To implement the parallel composition operator, <^>, we introduce another function, *pairs*, which takes two *initialized signals* as arguments and gives the initialized signal that emits the pairs of both arguments. The head of *pairs l r* is the pair of the head of *l* and the head of *r*. On each new emission of *l* or *r*, *pair l r* emits the pair of the current form of *l* and the current form of *r*. To achieve this, we first wait for the reactive computation of the tail of one of the initialized signals to complete and then update both initialized signals. This is done using the function *lup*: if the tail has not emitted a value yet, the initialized signal is the head of the old initialized signal followed by the new state of the computation of the tail. If the tail already emitted a value, the initialized signal is simply that tail. The function <^> is then implemented by first waiting for both signal computation to start emitting values, and then applying the second element to the first element of each pair.

4.5.3 Dynamic lists

The signal functions from the previous section can be used to define *dynList*, which takes a signal computation emitting initialized signals, and composes them in parallel. For this, we first define a dynamic variant of cons (:), that takes an initialized signal that has as form type something of type *a* (the head), and an initialized signal that emits something of type [*a*] (the tail) and returns the result of "consing" the head to the lists from the tail over time:

```
cons :: Ord e ⇒ ISig e a l → ISig e [a] r
              → ISig e [a] ()
cons h t = do (h, t) ← imap (uncurry (:)) (pairs h t)
              imap (:[ ]) h
              t
              return ()
```

The initialized signal *pairs h t* gives the pairs of the head and tail over time. Hence, if we transform these pairs so that the head is prepended to the tail, we get the list over time. After step *pairs h t*, either the head or the tail has ended. We then emit the residual values of the head and the tail, one of which is empty. In this way, if we are given two an initialized signals *a* and *b* of the same type and an initialized signal emitting lists of that type, *c*, then *a 'cons' (b 'cons' c)* will emit lists of the current states of *a*,*b* and *c*. If an an initialized signal ends, it has no current form and it will not be included in the list. For example, if *b* ends before *a* and *c*, then we will continue as *a 'cons' c*.

To define *dynList*, we start with the empty dynamic list, i.e. the initialized signal list that always has as current form the empty list. We then run this initialized signal until the argument of *dynList* emits a new initialized signal. Then, we prepend this new initialized signal to the current dynamic list to obtain the new dynamic list. Afterwards, we run this dynamic list until the argument emits another initialized signal and the process repeats.

```
dynList x = emitAll (idynList x)
idynList :: Ord e ⇒ Sig e (ISig e a l) r → ISig e [a] ()
idynList l = rl ([] :| hold) l ≫ return () where
  rl t (Sig es) = do (t, es) ← t 'iuntil' es
                     case es of
                       Done (e :| es) → rl (cons e t) es
                       _              → t
```

4.6 Comparison with other evaluation schemes

To implement reactive systems, one needs a basic mechanism to deal with events that occur over time. We identify four such mechanisms:

- Busy waiting
- Blocking I/O multiplexing
- Concurrency
- Callback networks

For each of these basic mechanisms there exists one or multiple corresponding FRP evaluation mechanisms. Our approach is the only one which uses blocking I/O multiplexing. In the following subsections we will discuss FRP evaluation schemes for each of these other basic mechanisms.

4.6.1 Busy waiting

The original FRP formulation [Elliott and Hudak, 1997] and Arrowized FRP [Courtney and Elliott, 2001] use an implementation which models signals as functions, which given an amount of time and input values return the pair of their current emission and their continuation function. Since in this approach signals do not communicate *which* events they are interested in, the *entire* signal expression must be evaluated on each update, including the parts for which the input did not change. The reactive interpreter does not know which events to wait for and is hence in a busy waiting loop, constantly calling the signal continuation function with the new time and possibly interesting event occurrences. Since this continuation-based implementation of signals is purely functional, it allow time-branching signals.

4.6.2 Concurrency

A second basic mechanism is to use *concurrency* in the form of multiple parallel threads or processes. Elliot [Elliott, 2009b] gives a FRP evaluation scheme which avoids unnecessary re-computations based on the following observation: if we know the order in which the events arrive *in advance*, then we could just use *blocking I/O* to implement FRP. He then introduces the concept of *unambiguous choice*: given two ways to compute the *same* value using blocking I/O, we can start both computations in parallel, see which one completes first, kill the other and use the result. This approach does not allow time-branching semantics, because the intermediate states of signals are simply not accessible as values.

4.6.3 Callback networks

The typical way to implement FRP using callbacks networks is to organize the system in a directed acyclic graph, where the nodes are signals and there is an edge between two signals if one depends on the other. Signals can then notify other signals if they update their value (i.e. emit). Variants of this basic model are used in many FRP systems, such as Scala.React [Maier and Odersky, 2012], FrTime for Racket [Cooper and Krishnamurthi, 2006], Frappe for Java [Courtney, 2001], and Microsoft's Reactive Extensions (Rx)[6].

As an example of such a network consider the following simple (Monadic) FRP expression:

```
let nrClicks = memo (scanl (+) 0 clicks)
in always (+) <^> nrClicks <^> filter isEven nrClicks
```

The dependency network of this expression is shown in Figure 4.9. As an example reduction, suppose that the current value of *nrClicks* is 3 and the value of *filter isEven nrClicks* is 2. Suppose then that the user presses a button, which will cause the signal *clicks* to update. This signal then calls *nrClicks*, which depends on it. The signal *nrClicks* then updates its value to 4 and calls the signals that depend on it. If it calls *filter isEven* first, then that also updates its value to 4 and calls +, which will then update its value to 8. However, if *nrClicks* calls + before *filter isEven*, then + will use a stale value of *filter isEven*, namely 2, and incorrectly update its value to 6.

[6]https://rx.codeplex.com/

$$\begin{tikzcd}
 & {} & \\
 & + \arrow[u] & \\
nrClicks \arrow[ur] & & filter\ isEven \arrow[ul] \arrow[ll] \\
clicks \arrow[u] & &
\end{tikzcd}$$

Figure 4.9: A simple signal dependency network.

An incorrect update due to the order of calls in the signal network, such as the update of + to 6, is called a *glitch*. Most FRP systems based on callback networks use a glitch prevention system. The exception is Rx, which does not prevent glitches (according to [Maier and Odersky, 2012]). The most common way to prevent glitches, is not to let signals call each other directly but instead place their calls in a priority queue [Cooper and Krishnamurthi, 2006; Maier and Odersky, 2012]. This priority queue schedules updates of nodes according to the topological ordering of the directed acyclic graph, which ensures that no glitches will occur.

A complication is then that the topology of the signal network may change dynamically and hence the system needs to maintain a topological ordering of the evolving directed acyclic graph. Another complication is that to prevent needless computations, we would like to prevent scheduling updates to signals that no other signal depends on. A non-solution is to use weak references for dependence links and then rely on the garbage collector to collect the dead signals. It may be a while before dead signals are collected, and during this time needless computations are possible. Hence there needs to be some form of instant garbage collection, for example reference counting. For more information on possible solutions for these complications see for example [Maier and Odersky, 2012; Cooper and Krishnamurthi, 2006] or [Courtney, 2001].

The difference between Monadic FRP and a glitch-free callback network based FRP system is that in Monadic FRP the events come in at the root of the expression and evaluation proceeds *top-down*, whereas in glitch-free callback network based FRP events arrive at the leaves and evaluation proceeds *bottom-up*. In Monadic FRP there is no way to create a glitch, as the expression itself *is* the ordering on signals. Since Monadic FRP traverses the signal network in top-down fashion, signals that no other signal depends on will never be computed, and are collected by ordinary garbage collection.

Another difference is that callback-based FRP systems use the signal network as *mutable data*, whereas in Monadic FRP the signal network is *immutable*, i.e. the next network is a new signal network, not a modification of the old network. This is the reason that time-branching operations are possible in purely functional evaluation models such as that of Monadic FRP and Arrowized FRP, and are impossible in callback-based systems.

4.7 Conclusion and Future work

In this chapter we introduced Monadic Functional Reactive Programming, an alternative programming model and evaluation mechanism for FRP. The basic notion in Monadic FRP is a reactive computation, a monadic computation which may require the occurrence of external events to continue. A signal computation is a reactive computation that may also emit values. In contrast to signals in other FRP formulations, signal computations can end. This leads to a monadic interface for sequencing signal phases, which is arguably more intuitive and flexible than the switching combinators found in other FRP approaches. This also allows us to define dynamic lists, lists that change over time, more easily than in other FRP approaches. In contrast to other FRP approaches, Monadic FRP can be implemented straightforwardly in a *purely functional* way while preventing redundant re-computations.

This gives rise to several directions for further research:

- How can mutually depended signals be expressed in Monadic FRP?
- Arrowized FRP does not require manual invocations of memoization functions like Monadic FRP and makes it possible to define mutually dependent signals. We are currently investigating whether it is possible to combine Monadic FRP and Arrowized FRP into a single framework that has the best of both worlds.
- How can Monadic FRP be formulated in a dependently typed setting, allowing us to statically rule out more meaningless and incorrect programs in the style of Sculthorpe and Nilsson [Sculthorpe and Nilsson, 2009]?
- How can Monadic FRP be integrated with a declarative graphics library, such as our previous work (Chapter 2)?

5

Reflection without Remorse

Revealing a hidden sequence to speed up monadic reflection[1]

Summary

A series of list appends or monadic binds for many monads performs algorithmically worse when left-associated. Continuation-passing style (CPS) is well-known to cure this severe dependence of performance on the association pattern. The advantage of CPS dwindles or disappears if we have to examine or modify the intermediate result of a series of appends or binds, before continuing the series. Such examination is frequently needed, for example, to control search in non-determinism monads.

We present an alternative approach that is just as general as CPS but more robust: it makes series of binds and other such operations efficient regardless of the association pattern – and also provides efficient access to intermediate results. The key is to represent such a conceptual sequence as an efficient sequence data structure. Efficient sequence data structures from the literature are homogeneous and cannot be applied as they are in a type-safe way to series of monadic binds. We generalize them to *type aligned sequences* and show how to construct their (assuredly order-preserving) implementations. We demonstrate that our solution solves previously undocumented, severe performance problems in iteratees, LogicT transformers, free monads and extensible effects.

5.1 Introduction

It is well-known that list-concatenation (++) is not efficient when its left argument is itself the result of a concatenation. A popular solution to this problem is to use continuation-passing style in the form of difference lists. We recall the problems

[1]This chapter was published earlier as: A. van der Ploeg, O. Kiselyov. Reflection without remorse: revealing a hidden sequence to speed up monadic reflection. In *Proceedings of the '14 Symposium on Haskell*, pages 133–144, 2014.

of list-concatenation and how continuation-passing style remedies it in Sections 2 and 3 respectively. However, continuation-passing style only solves the performance problem for certain usage patterns: if we need to observe intermediate results of concatenations, or build concatenations with sub-lists of other concatenations, then performance quickly degenerates. In other words: continuation-passing style again leads to performance problems if we alternate between building and observing.

In this chapter, we show that this pattern also occurs in many other situations, which at first blush have nothing to do with lists. In many implementations of monads (e.g., iteratees and non-determinism monads), a series of binds ($\gg=$) or choices (mplus), is quite like a series of list appends: they perform badly when left-associated. Like with lists, continuation-passing style makes such series perform algorithmically well regardless of the association pattern [Voigtländer, 2008]. However, several monads also support *monadic reflection* [Filinski, 1994], a way to observe and modify (a representation of) the current state of the computation. For example, the current state of a non-deterministic computation may be observed as a stream of results. We may remove the top result and continue with the rest – which is exactly what is needed to implement committed choice [Kiselyov et al., 2005]. Such monadic reflection destroys the performance advantage of the continuation-passing style. This chapter shows that one does not have to regret reflection.

For lists, the solution to the append-and-observe problem is to use a more suited sequence data structure, i.e. one that supports both head/tail and append operations efficiently. Such data structures can give an asymptotic improvement over both regular lists and difference lists. The surprise of this chapter is that such efficient data structures can also give an asymptotic improvement for other problematic occurrences of the build-and-observe pattern, in particular, monads and monadic reflection. The key insight is that we can reveal the hidden, abstract sequence of monadic binds: we can represent it as a concrete sequence. By then choosing the most suited sequence data structure for the problem at hand, performance can be greatly improved.

However, the literature on efficient sequences deals with homogeneous collections. In a 'sequence' of binds, the types of the 'elements' may vary. To solve this problem, we introduce a generalization of sequences called *type aligned sequences*: heterogeneous sequences where the types enforce the element order. In this way, we can solve the performance problem in any situation exhibiting the problematic pattern, in a completely type-safe way.

We were confronted with the performance problems of monadic reflection in projects using monadic functional reactive programming [van der Ploeg, 2013] and the parallel composition of iteratees [Kiselyov, 2012]. These practical problems have motivated the present research. We have distilled the issue into a performance problem with simple tree substitutions, which helped us see how changing the data representation to use efficient sequences can improve performance. This not only solves the original problem, but also gives a drop-in replacement for free monads [Swierstra, 2008] with better performance characteristics than previous approaches: examining a free monad value and binding it are both efficient, letting us alternate between these operations without performance penalty. This improved free monad leads, among other things, to an implementation of extensible effects [Kiselyov et al., 2013] in which a wider range of effects can be modeled efficiently.

We begin with some background: Section 2 recalls the problematic build-and-observe pattern in several guises, and we discuss continuation passing style and its performance problems in Section 3. Then we present our contributions:

- We present a solution to the build-and-observe problem for any monoid where left-associated expressions are more costly than right-associated expressions, giving an asymptotic running time improvement over both direct and continuation-passing style. (Section 4)
- We generalize our solution for monoids to monads, making left-associated bind expressions as well as monadic reflection efficient. (Section 4)
- We introduce type aligned sequences. As an example, we show an implementation of efficient type aligned queues. (Section 5)
- We show how our method solves previously undocumented, severe performance problems with monadic reflection in iteratees, LogicT transformers, free monads and extensible effects. (Section 6)

And in Section 7 we conclude.

The code accompanying this chapter is available at:
`https://github.com/atzeus/reflectionwithoutremorse`
The code in this chapter is in Haskell, but our approach can be used in any language with GADTs (indexed data types).

5.2 The problematic pattern and its cost

In this background section we recall the performance problems of associative operators that traverse their left argument but not their right argument. In particular, we discuss list concatenation, tree substitution and generic tree substitution. We recall that the running time cost of equivalent expressions involving such operators can differ asymptotically.

5.2.1 A first example: list concatenation

To analyze the performance problems of list concatenation, we recall the relevant standard definitions:

```
data [a] = [] | a : [a]

[]      ++ r = r
(h : t) ++ r = h : t ++ r
```

To append two lists, we must traverse all elements of the first list to arrive at the empty constructor at the end. Hence, reducing $x ++ y$ to normal form requires $|x| + 1$ case distinctions, from now on called steps, where $|x|$ is the length of x.

One might argue that this is not a problem: thanks to laziness, observing the head of x ++ y is just observing the head of x, plus one extra step. To observe the n-th element of a list we must traverse the list anyway: concatenation just adds one extra step per element.

The real problem arises if the left argument is itself the result of a concatenation. For example, in the expression (x++y)++z, the list x must be traversed *twice*: it occurs twice in a left hand side argument to ++. Hence, this expression runs in in $2|x|+|y|+2$ steps, whereas the *equivalent* expression x ++ (y ++ z) runs in just $|x| + |y| + 2$ steps. In this way, a wrong grouping of expressions involving ++ can easily lead to severe performance problems, as we shall see in full generality in §5.2.4.

5.2.2 Another example: Tree substitution

A different guise of the same problem occurs with trees and an operation which substitutes the leaves of a tree with another tree:

```
data Tree =  Node Tree Tree
          |  Leaf

(↩) :: Tree → Tree → Tree
Leaf       ↩ y = y
(Node l r) ↩ y = Node (l ↩ y) (r ↩ y)
```

The performance situation is similar: evaluating (x ↩ y) ↩ z traverses x twice, whereas the equivalent x ↩ (y ↩ z) only traverses x once. Hence evaluating the former expression costs $|x|$ steps more than evaluating the latter, where |x| is now the number of inner nodes in x.

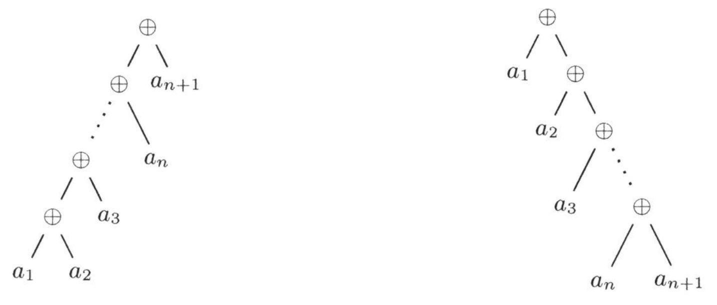
For lists, this problem can be solved by simply using a catenable (meaning with fast concatenation) sequence data structure instead of a regular head-tail list. For trees, the solution is not so obvious. Should we investigate a new specialized data structure for trees or browse the literature to see if someone else has already invented it? (Hint: No.)

5.2.3 A Monadic example: Generic trees

The performance degradation from a bad association occurs not only with monoids, such as lists and trees. If we generalize our tree to a generic tree, with data at the leaves, then substitution becomes the monadic bind (>>=)[2]:

```
data Tree a =  Node (Tree a) (Tree a)
            |  Leaf a

(↩) :: Tree a → (a → Tree b) → Tree b
(Leaf x)   ↩ f = f x
(Node l r) ↩ f = Node (l ↩ f) (r ↩ f)
```

[2]This example is taken from [Voigtländer, 2008].



(a) A left-associated expression

(b) A right-associated expression

Figure 5.1: Equivalent left- and right-associated expressions.

```
instance Monad Tree where
  return = Leaf
  (≫=)   = (↩)
```

The performance situation is obviously the same: the only thing that changed is that $\hookleftarrow$ now takes a function as its right argument. Although $\hookleftarrow$ and $\gg=$ are not associative operators in the strict sense, they satisfy the similar associativity monad law:

$$(\mathsf{m} \gg= \mathsf{f}) \gg= \mathsf{g} \quad \equiv \mathsf{m} \gg= (\lambda \mathsf{x} \rightarrow \mathsf{f}\ \mathsf{x} \gg= \mathsf{g})$$

We now see that the situation is the same: $(\mathsf{m} \gg= \mathsf{f}) \gg= g$ runs in $|\mathsf{m}|$ steps more than the equivalent $\mathsf{m} \gg= (\lambda\, x \rightarrow \mathsf{f}\ x \gg= g)$.

Note that while bind is not strictly an associative operator, the following operator, known as Kleisli composition, is strictly an associative operator:

```
(⋙) :: Monad m ⇒ (a → m b) → (b → m c) → (a → m c)
f ⋙ g = λx → f x ≫= g
```

The similarity with the situation with lists and non-generic trees can then be made even stronger: $(\mathsf{p} \ggg \mathsf{q}) \ggg \mathsf{r}$ is more costly than the equivalent $\mathsf{p} \ggg (\mathsf{q} \ggg \mathsf{r})$.

5.2.4 Asymptotic running time overhead

In general, the problem occurs with any *associative* (or satisfying the associativity monad law) operator that traverses its left argument but not its right argument that operates on some *recursive*[3] data type. In this situation, $(\mathsf{x} \oplus \mathsf{y}) \oplus \mathsf{z}$ costs $|\mathsf{x}|$ more

[3]If the data type is not recursive, e.g., the Maybe monad, one can easily see that both left and right associations have the same asymptotic cost.

steps to evaluate than $\mathsf{x} \oplus (\mathsf{y} \oplus \mathsf{z})$, where $|\mathsf{x}|$ is now the number of values of type X inside x that are non-terminal (i.e. they are not for example the empty list or a leaf).

Repeated application of such an operator can lead to asymptotic running time overhead if $|\mathsf{a} \oplus \mathsf{b}| \geqslant |\mathsf{a}| + |\mathsf{b}|$. For lists, this obviously holds since $|\mathsf{a} \mathbin{+\!\!+} \mathsf{b}| = |\mathsf{a}| + |\mathsf{b}|$. For trees, the size of $\mathsf{a} \hookleftarrow \mathsf{b}$ is $|\mathsf{a}| + \mathsf{a}_l|\mathsf{b}|$, where a_l is the number of leaves in the tree a. Since there is at least one leaf in a tree, the inequality $|\mathsf{a} \hookleftarrow \mathsf{b}| \geqslant |\mathsf{a}| + |\mathsf{b}|$ holds.

That this leads to asympotic running time overhead can be seen as follows: a *left-associated expression*, as visualized in Figure 5.1(a):

$$(((a_1 \oplus a_2) \oplus a_3) \cdots \oplus a_n) \oplus a_{n+1}$$

then costs at least $\sum_{i=1}^{n-1}(n-i)|a_i|$ more steps than the equivalent *right-associated expression*, visualized in Figure 5.1(b):

$$a_1 \oplus (a_2 \oplus (a_3 \oplus \ldots (a_n \oplus a_{n+1}) \ldots))$$

If we assume that all elements have size one, i.e. $|a_i| = 1$, then we more easily see that a left-associated expression costs $O(n^2)$ more steps than a right-associated expression:

$$\sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

Of course, these are the most extreme cases: most expressions will not be completely right- or left-associated. However, any expression that is not completely right-associated will yield an overhead. We cannot expect the programmer to only form right-associated expressions, especially when using laziness: the programmer must then make sure that every time the operator is used, the left hand side *cannot* be itself a result of this operator.

5.3 Continuation-passing style

In this second background section, we discuss a popular way to alleviate such performance problems for certain usage patterns, namely *continuation-passing style.* We illustrate this technique with difference lists, which use continuation-passing style to speed up list concatenation. We then show that difference lists only avoid performance problems if we do not alternate between building and observing and that the same holds for continuation-passing style in general.

5.3.1 Difference lists

The trick of difference lists [Hughes, 1986] is to *only* build right-associated expressions. More precisely, difference lists are *functions* for building right-associated expressions, i.e. functions of the form:

$$\lambda\, t \ \rightarrow a_1 \mathbin{+\!\!+} (a_2 \mathbin{+\!\!+} (a_3 \mathbin{+\!\!+} (a_4 \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} t)))$$

And hence we define difference lists as functions from lists to lists:

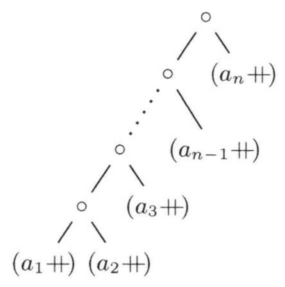


Figure 5.2: Difference list with worst case conversion characteristics.

```
type DiffList a = [a] → [a]
```

We can convert a difference list to a regular list by simply feeding it the empty list:

```
abs :: DiffList a → [a]
abs a = a []
```

To convert a list to a difference list, we partially apply ++:

```
rep :: [a] → DiffList a
rep = (++)
```

Concatenation is then simply function composition, since (a ++) ∘ (b ++) ≡ λt → a ++ (b ++ t)[4] :

```
(+̂+) :: DiffList a → DiffList a → DiffList a
(+̂+) = (∘)
```

The trick is then to concatenate using difference lists, and then convert the result to a list when needed. Since this will always produce a right-associated expression, the overhead associated with expressions that are not right-associated is avoided.

However, the problem with this technique is that converting a list to a difference list is expensive in the long run. Conversion of a list l to a difference list is simply (l ++), which, when the final result is observed, contributes the costs of |l| steps, adding one operation to each node in the list. Hence, if we convert back and forth n times, this will cost n|l| steps. Of course, converting the same list back and forth a number of times is a bit of a contrived situation. However, the problem also occurs if we convert a difference list to a list and convert *part* of the list back to a difference list.

Another, more subtle problem is that conversion in the other direction, from a difference list to a list, is not a constant time operation. We cannot *observe* anything directly on a difference list, for example we cannot see whether it is empty, and hence conversion to a regular list is often required. This conversion is not cheap: in the

[4]We use the notation (x ++) as a shorthand for (λy → x ++ y).

worst case the difference list consists of a left-associated expression of the following form, which is visualized in Figure 5.2:

$$((((a_1 +\!\!+) \circ (a_2 +\!\!+)) \circ (a_3 +\!\!+)) \ldots +\!\!+ (a_{n-1} +\!\!+)) \circ (a_n +\!\!+)$$

Converting such a difference list to list, by applying [] to it, then requires n invocations of $\circ$ to reduce to the following list expression:

$$a_0 +\!\!+ (a_1 +\!\!+ (a_2 +\!\!+ (a_3 +\!\!+ \ldots +\!\!+ (a_n +\!\!+ []))))$$

Only after these operations we can reduce further and inspect the resulting list to see whether it is empty or not. Hence, observing (parts of) intermediate lists can also lead to performance problems.

To summarize: difference lists only solve performance problems if our usage of lists is strictly separated into a build (i.e. concatenation) phase and an observation phase. If we alternate between building and observing, as is often needed, then performance problems will resurface.

5.3.2 General Continuation-passing style

The trick of difference lists, i.e. continuation-passing style, can be applied in many situations. For example, it can be applied to any monoid[5]:

```
type DiffMonoid a = a → a
abs :: Monoid a ⇒ DiffMonoid a → a
abs a = a mzero
rep :: Monoid a ⇒ a → DiffMonoid a
rep = mappend
instance Monoid a ⇒ Monoid (DiffMonoid a) where
  mempty   = id
  mappend  = (∘)
```

If we apply the trick to monads, we get the codensity monad transformer [Jaskelioff, 2009], which is highly related to the continuation monad [Liang et al., 1995]:

```
type CodensityT m a = ∀ b. (a → m b) → m b
abs :: Monad m ⇒ CodensityT m a → m a
abs a = a return
rep :: Monad m ⇒ m a → CodensityT m a
rep = (>>=)
instance Monad m ⇒ Monad (CodensityT m) where
  return a = rep (return a)
    -- or equivalently: λk → k a
  m >>= f = m ∘ flip f
    -- or equivalently: λk → m (λa → f a k)
```

[5]To reduce clutter, we ignore the fact that DiffMonoid and CodensityT should actually be a **newtype** in Haskell.

The codensity monad transformer is often used for solving the performance problems of left-associated expressions [Claessen, 2004; Voigtländer, 2008]. As with difference lists, this works fine if our usage is separated in a build and an observations phase. However, if we have another usage pattern, alternating between building and observing, the same problems as with difference lists occurs: continuation-passing style reintroduces performance problems.

5.4 Solving the problem

The main insight for our solution is that expressions of the form:

$$a_0 \oplus a_1 \oplus a_2 \oplus \cdots \oplus a_n$$

are sequences and that such abstract sequences should be represented *explicitly*. With the previous approaches such sequences are only represented *implicitly*. More precisely, when directly using $\oplus$, these sequences are implicitly represented at runtime as trees where the leaves are the elements and nodes are (delayed) function applications. When using continuation-passing style, such sequences are also represented as trees, but now the leaves are functions representing the elements and the nodes are function composition. By making representation of these sequences explicit, we can choose a more suited sequence data structure and performance problems can be solved for any usage pattern.

We first illustrate our solution by applying it to tree substitution. We then show that applying our solution to generic trees requires type aligned sequences and how such type aligned sequences can be used to solve the problem. Afterwards, we discuss the general solution.

5.4.1 A first example: tree substitution

We want to replace the implementation of the Tree data type and the substitution operator such that they have the same semantics, but better performance characteristics. Hence we will redefine the following operations:

- Observing a tree, i.e. viewing if it is a leaf or node.
- Constructing a leaf or node.
- The leaf substitution operator.

We are not concerned with other operations on trees here, they are defined in terms of the above operations.

Before we define our new data type Tree', let us start with defining what the result of observing a tree should be. Analogous to viewing a sequence data structure from the left or right, we can view a tree by observing if its root node is a leaf or a node:

```
data TreeView = Node Tree' Tree'
              | Leaf
```

Notice that the children of a Node are *not* of type TreeView, they are of the new (yet to be defined) Tree' type. To pattern match on a value of type Tree', we first need to call a function that gives the view of the Tree', i.e. a function of type:

```
toView :: Tree' → TreeView
```

This pattern is common in data abstraction [Wadler, 1987]: it allows us to hide the implementation of the Tree' type, while still being able to pattern match on it. It is, for example, also used in efficient sequence data structures, such as the one in Data.Sequence: the pattern is used to hide the implementation of the sequence such that the user cannot differentiate between things which have multiple representation, but have the same meaning.

The Glasgow Haskell Compiler has a syntactic extension called *view patterns* which eases the usage of such data types. More precisely, it allows us to apply such a view function inside a pattern match. As an example of this, with our previous tree data type we could write a function:

```
isLeaf Leaf = True
isLeaf _    = False
```

With view patterns, this function on the new Tree' type becomes:

```
isLeaf (toView → Leaf) = True
isLeaf _               = False
```

In this way, the syntactic inconvenience of our technique is minimized.

The implementation of the Tree' data type is an explicit expression: a sequence of trees $a_0, a_1, \ldots, a_n$, such that that the result of observing such a Tree' is $a_0 \hookleftarrow a_1 \hookleftarrow \ldots \hookleftarrow a_n$.

```
newtype Tree' = Tree' (CQueue TreeView)
```

Where CQueue is an efficient sequence data structure, which we assume to be an instance of the type class for sequences defined in Figure 5.3(a). Very efficient purely functional sequence data structures exist: data structures where both concatenation and head/tail access run in amortized constant time [Okasaki, 1995], and even data structures where both run in worst case constant time [Kaplan and Tarjan, 1999b; Okasaki, 1995].

The elements of the sequence are of type TreeView, which is mutually recursive with Tree': the children of the elements in the expression are again explicit expressions. The Tree' type is a **newtype** instead of a type alias, such that we can omit the Tree' constructor from the interface, making Tree' an *abstract type*.

Constructing a leaf or node of type Tree' is then done by converting a TreeView value to a Tree' by using the following function:

```
fromView :: TreeView → Tree'
fromView x = Tree' $ singleton x
```

The resulting tree is not (yet) an argument to the substitution operator and hence it is represented as a sequence of length one. Notice that fromView is the inverse of toView.

The implementation of the substitution operator $\hookleftarrow$ is then simply to concatenate the two explicit expressions:

```
(↩) :: Tree' → Tree' → Tree'
(Tree' l) ↩ (Tree' r) = Tree' (l ⋈̂ r)
```

Since we are using an efficient sequence data structure, this concatenation only takes (amortized) constant time.

The implementation of $\hookleftarrow$ no longer defines how to actually replace the leaves of a tree with another tree. Instead this logic is moved to the toView function, which converts an explicit expression to its view (i.e. its head normal form).

```
toView :: Tree' → TreeView
toView (Tree' s) = case viewl s of
  EmptyL → Leaf
  h ◁̂ t → case h of
       Leaf      → toView (Tree' t)
       Node l r → Node (l ↩̂ t) (r ↩̂ t)
  where (↩̂) :: Tree' → CQueue TreeView → Tree'
        (Tree' l) ↩̂ r = Tree' (l ⋈̂ r)
```

Where viewl is a function that allows us to view the sequence from the left: see if it is empty or obtain the head and tail. In contrast to continuation-passing style, converting an explicitly represented expression to an observable value does not mean converting the entire explicitly represented expression: we partially convert, keeping the children of a node as explicit expressions.

In this way, all operations we want to support, namely construction, observation and substitution have become efficient operations. Moreover, the expressions , $(x \hookleftarrow y) \hookleftarrow z$ and $x \hookleftarrow (y \hookleftarrow z)$ lead to the same sequence, and hence performance does not depend on the association pattern. It should hence come as no surprise that this approach also solves performance problems if we alternate between building trees using substitution and observing the result of such substitutions.

5.4.2 Solving the performance problems of generic trees using type aligned sequences

But what if we want to apply our solution to generic trees? We must then explicitly represent expressions of the form:

$$m \ggg= f_1 \ggg= f_2 \ggg= f_3 \ldots \ggg= f_n$$

The problem is that each f_i has type a → Tree b, for some a and b, and these types can *differ* between elements. This means we *cannot* use a regular sequence: to use it all elements must be of the same type.

To be able to apply our solution to such situations, we generalize sequences to *type aligned sequences*: sequences parametrized by a type constructor c, such that each element is of type c a b, for some a and b. If the last type argument to c of an element is a, then first type argument to c in the next element (if any) *must be* a. If

```
class Sequence s where
  empty       :: s a
  singleton   :: a → s a
  (⋈̂)         :: s a → s a → s a
  viewl       :: s a → ViewL s a

data ViewL s a where
  EmptyL      :: ViewL s a
  (◁̂)         :: a → s a → ViewL s a
```

(a) A type class for regular sequences.

```
class TSequence s where
  tempty     :: s c x x
  tsingleton :: c x y → s c x y
  (⋈)        :: s c x y → s c y z  → s c x z
  tviewl     :: s c x y → TViewl s c x y

data TViewl s c x y where
   TEmptyL  :: TViewl s c x x
   (◁)      :: c x y → s c y z → TViewl s c x z
```

(b) A type class for type aligned sequences.

Figure 5.3: Type classes for type aligned and regular sequences.

we set the type constructor c to (→), we get type aligned sequences of functions: the output type of a function is then always the input type to the next function.

In the next section we discuss such type aligned sequences in depth and show they can be defined. For now, let us assume that we have an efficient type aligned sequence data structure called TCQueue, which is an instance of the type aligned sequence type class defined in Figure 5.3(b).

The elements in the sequence described above are of type a → Tree' b, for some a and b, except the first element m. We need a type constructor to describe this pattern:

```
type TreeCont a b = a → Tree' b
```

A type aligned sequence where each element is a TreeCont is then of the following type[6]:

```
type TreeCExp a b = TCQueue TreeCont a b
```

The situation is now a bit different than with our non-generic trees: an expression involving a series of binds must always start with an element of type Tree' a, whereas the rest of the elements are of type TreeCont a b, for some a and b. Hence, we implement the tree data type as explicit expression containing a first element and a sequence of right-hand-side arguments to bind.

```
data Tree' a where
  Tree' :: TreeView x → TreeCExp x a → Tree' a

data TreeView a = Leaf a  |  Node (Tree' a) (Tree' a)
```

This definition uses an existential type x: the first element in the expression may be a tree of any type, as long as the result of the expression is a tree containing elements of type a.

The fromView and ↩ functions are adapted accordingly:

```
fromView :: TreeView a → Tree' a
fromView x = Tree' x tempty

(↩) :: Tree' a → (a → Tree' b) → Tree' b
(Tree' x s) ↩ f = Tree' x (s ⋈ tsingleton f)
```

As before, the actual logic of substitution is moved to the view function:

```
toView :: Tree' a → TreeView a
toView (Tree' b t) = case b of
   Leaf a → case tviewl t of
     TEmptyL → Leaf a
     h ◁ t    → toView ((h a) ↩̂ t)
   Node l r → Node (l ↩̂ t) (r ↩̂ t)
 where (↩̂) :: Tree a → TreeCExp a b → Tree b
       (Tree' b l) ↩̂ r = Tree' b (l ⋈ r)
```

[6]To reduce clutter, we ignore that TreeCont must be a **newtype** for this to work in current Haskell.

In this way, the performance problems for any usage pattern of generic trees have also disappeared by using type aligned sequences.

5.4.3 The general case

Suppose we have some recursive data type X and an associative operator traversing its left argument but not its right argument. The solution is then to replace the data type X by an abstract data type X' and rewrite the problematic operator by performing the following steps:

1. Replace X with two mutually recursive data types: one for the abstract type containing the explicit expression (X') and one view type, which is the same as the original X, but the self-references have been replaced by X'.

2. Define the original operator on X' by concatenating the explicit expressions.

3. Define a fromView function that converts a view value to an X' expression by constructing an explicit expression with one element.

4. Define a toView view function that evaluates an explicit expression to its view, using the workings of the original operator.

A type aligned sequence must be used if the type of the right argument of the operator depends on the type of the left argument of the operator.

Notice that explicitly representing expressions in this way means that applying the operator with the identity element does not necessarily immediately yield the original value. For example, m $\gg=$ return and m are different *expressions*. However, we cannot observe this difference by viewing m $\gg=$ return and m. Hence, the identity element is an identity element *up to observation*. Associativity laws directly hold, since sequence concatenation is associative. To ensure that we do not accidentally differentiate between m $\gg=$ return and m, it is important to define the result of the above steps in an separate module and to not export the constructor of X'.

This process gives an abstract type X', with operations to construct, observe (view) and apply the operator. We argue that this resulting data type X' has the same semantics as the original data type, provided that X' is abstract. We feel that a formalization of these steps and a proof of the isomorphism of X and X' should be possible, but it is beyond the scope of this chapter.

5.5 Type aligned sequences

In the previous section, we saw that type aligned sequences are required to explicitly represent expressions involving operators where the type of the left argument depends on the type of the right argument. We now introduce type aligned sequences, discuss their relation with regular sequences, and show an example of how a sequence data type can be converted into a type aligned sequence data type.

5.5.1 Definition and intuition

Type aligned sequences are best explained by an example: a type aligned sequence of *functions* is a sequence $f_1, f_2, f_3 \ldots f_n$ such that the composition of these functions $f_1 \circ f_2 \circ f_3 \circ \ldots \circ f_n$ is well typed. In other words: the result type of each function in the sequence must be the same as the argument type of the next function (if any). In general, the elements of a type aligned sequence do not have to be functions, i.e. values of type $a \to b$, but can be values of type (c a b), for some binary type constructor c. Hence, we define a *type aligned sequence* to be a sequence of elements of the type (c $a_i b_i$) with the side-condition $b_{i-1} = a_i$. If s is the type of a type aligned sequence data structure, then (s c a b) is the type of a type aligned sequence where the first element has type (c a x), for some x, and the last element has type (c y b), for some y.

It may be instructive to think of a type aligned sequence as a *path through a directed graph.* In this directed graph each node is a *type* and there is an edge from type a to type b *for each value* of type (c a b). Hence, we call a value of type (c a b) a c-*edge*. A type aligned sequence of type (s c a b) is then a sequence of c-edges such that they form a path from a to b trough this graph: the target of each edge is the source of the next edge.

Type aligned sequences can be defined using Generalized Algebraic Data Types (GADTs) [Nilsson, 2005]. As a simple example of this, consider a type aligned list:

```
data TList c x y where
  Nil  :: TList c x x
  (◦:) :: c x y → TList c y z → TList c x z
```

In the graph interpretation, the empty type aligned sequence corresponds to an empty path, and hence the empty list is a path from x to x, for any x. The Cons constructor adds one c-edge to the front of a path, the types ensure that the target of this c-edge is the source of the rest the path.

5.5.2 Relation with regular sequences

The only difference between regular sequences and type aligned sequences are the types: TList differs from the ordinary list only in the more precise types of its constructors. In fact, type aligned sequences are a *generalization* of regular sequences: any type aligned sequence can be used as a regular sequence, but not the other way around. We can use a type aligned sequence as a regular sequence by effectively "partially erasing" the extra types with the following construction:

```
data AsUnitLoop a b c where UL :: a → AsUnitLoop a () ()
```

By using this construction, there exists an edge from () to () for each value of type a in the graph interpretation. Since there are no other edges, the graph effectively has just one node: the other types are unreachable. Hence, a regular list $a_1 : a_2 : a_3 \ldots a_n : []$ of type [a] corresponds to a type aligned list:

$$\mathsf{UL}\ a_1 \mathbin{\overset{\circ}{:}} \mathsf{UL}\ a_2 \mathbin{\overset{\circ}{:}} \mathsf{UL}\ a_3 \ldots \mathsf{UL}\ a_n \mathbin{\overset{\circ}{:}} \mathsf{Nil}$$

of type TList (AsUnitLoop a) () () . This type aligned list corresponds to a path of length n through the graph consisting solely of self-loops on (), where each edge corresponds to a value of type a.

We can use this construction to provide an instance for the regular sequence class (Figure 5.3(a)) for any instance of the type aligned sequence class (Figure 5.3(b)):

```
type AsSequence s a = s (AsUnitLoop a) () ()

instance TSequence s ⇒ Sequence (AsSequence s) where
  empty      = tempty
  singleton  = tsingleton ∘ UL
  (⋈̂)        = (⋈)
  viewl  s   = case tviewl s of
                    EmptyL     → TEmptyL
                    UL h ◁ t   → h ◁̂ t
```

A benefit of using type aligned sequences in this way, instead of directly using regular sequences, is that type aligned sequences rule out a class of implementation bugs: the types in a type aligned sequence enforce the ordering of the elements. Hence, accidentally switching two elements will result in a type error, as the resulting sequence may not be a path. In contrast, in regular sequences the types do not enforce the ordering of the elements and an accidental change of order in, for instance, the definition of concatenation would have gone unnoticed by the type checker.

In general, sequences, i.e. words over some alphabet, are *free monoids*, whereas paths through a directed graph are *free categories* [Awodey, 2006]. Sequences in programming languages typically are homogeneous: they require that each element has the same type. The alphabet is then the set of values of the given type. Similarly, type aligned sequences are paths through the directed graph where the edges are formed by the values of type (c a b), for all types a and b.

Indeed, any sequence data type can be made an instance of Monoid, without assuming anything about the elements of the sequence. Similarly, any type aligned sequence data type can be made an instance of Category, without assuming anything about the elements of the type aligned sequence:

```
instance Sequence s ⇒ Monoid (s a) where
  mempty  = empty
  mappend = (⋈̂)

instance TSequence s ⇒ Category (s c) where
  id  = tempty
  (∘) = flip (⋈)
```

The fact that we can use any type aligned sequence as a regular sequence also has a theoretical motivation: a monoid corresponds to a category with just one object, the elements in the monoid are now arrows (morphisms) from this one object to itself and the monoid operation is arrow composition [Awodey, 2006]. Hence, a free monoid corresponds to the free category over a graph with just one node, where the

```
data Pair c a b where
  (×) :: c a w → c w b → Pair c a b

data Buffer c a b where
  B1 :: c a b → Buffer c a b
  B2 :: Pair c a b  → Buffer c a b

data Queue c a b where
  Q0 :: Queue c a a
  Q1 :: c a b → Queue c a b
  QN :: Buffer c a x → Queue (Pair c) x y
        → Buffer c y b → Queue c a b

(|▷) :: Queue c a w → c w b → Queue c a b
q |▷ b = ...
viewl :: Queue c a b → TViewl Queue c a b
viewl q = ...
```

Figure 5.4: A type aligned queue data structure.

self-edges correspond to the elements of the alphabet. This is exactly what we did with AsUnitLoop above: it makes every value of type a into a self-edge on the node ().

5.5.3 An example of making sequences type aligned: efficient queues

Generalizing the types of a sequence data type so that it becomes a type aligned sequence data type, means generalizing the constructor types, and assuring (that is, "proving" to the type checker) that all operations on the data type preserve the element order. This generalization requires some creativity but in our experience, it is a straightforward operation. In the code accompanying this chapter we show type aligned versions of *finger trees* [Hinze and Paterson, 2006] and of a worst case constant time catenable queue [Okasaki, 1995, 1998].

As an not entirely trivial example of turning a sequence data structure into a type aligned sequence data structure, consider the (non-catenable) queue shown in Figure 5.4. This data structure is essentially the same as the queue presented in Okasaki's *Purely functional Data Structures* [Okasaki, 1998, §8.4] but the types have been generalized.

To generalize this queue to a type aligned sequence data structure, we needed to generalize not only the types of the constructors of the queue, but also the types of the constructors of the pairs and buffers of which it consists. Before generalizing the types, both elements of a pair had the same type, but now the elements are c-edges such that they form a path of length two. A buffer can hold either a single element or a pair and the types of these constructors have been generalized straightforwardly. Slightly less obvious is generalizing the types of the constructors of a queue. A queue may consist of nested queues: if a queue has more than one element (constructor QN), it is represented as two buffers and a *queue of pairs*. With generalized types, the type of this queue of pairs is a type aligned queue holding (Pair c)-edges, i.e. paths of length two.

The only difference in the operations, namely en-queuing and viewing the head-/tail, is their type signatures, the operations themselves are left unchanged and are hence not shown. The full code for these type aligned queues is included in the code accompanying this chapter.

5.6 Fast Monadic Reflection

In this section we show how our solution can be used in various real-life monads. In particular, several monads offer *monadic reflection*: a way to observe, or reify, the internal state of the computation, represented in a suitable data structure. For example, the internal state of a non-determinism monad can be observed as a stream of choices. This terminology is due to Filinski [Filinski, 1994] who modeled it after the terminology of Wand and Friedman [Friedman, 1988]. Monadic reflection leads to alternating between building and observing, and hence leads to previously undocumented, severe performance problems. We demonstrate several examples of how we can factor out sequences in monads such that monadic reflection can be efficiently supported. In particular, we discuss LogicT transformers, iteratees (and related constructs), free monads and extensible effects.

5.6.1 LogicT Monad Transformers

As a first example of how we can apply our solution to a practical example, consider non-determinism monads. The MonadPlus type class extends the Monad interface with support for non-deterministic choice with backtracking. The most obvious instance of this interface is the list monad: bind is then concatMap (with the order of the arguments reversed) and mplus is concatenation. The usage of list concatenation can lead to performance problems, which can be solved by simply using a catenable queue instead.

Kiselyov, Shan, Friedman and Sabry [Kiselyov et al., 2005] showed that a large class of logical effects, namely cut, soft cut, interleaving and fair conjunction, can all be expressed when a single function is added to the interface. This function, called msplit, essentially splits the logical computation into a computation of the first result and computation of the rest of the results. More precisely, this function has type:

```
class MonadPlus m ⇒ MonadLogic m where
  msplit :: m a → m (Maybe (a, m a))
```

It takes a logical computation and turns it into another logical computation, namely one which returns Nothing if the original logical computation had no results, and otherwise returns a Just value carrying a tuple of the first result and the logical computation of the rest of the results. This is an instance of monadic reflection: msplit allows us to observe the internal state of the monad as a stream of results. The implementation of this msplit function for lists and other sequence data structures is straightforward: it converts the empty sequence to Nothing and a non-empty sequence to a Just value of the head and tail.

However, an efficient monad *transformer* that adds non-determinism to an arbitrary monad is not defined so easily. In a functional pearl [Hinze, 2000], Hinze systematically derives such a non-determinism monad transformer implementation. He then notes that a left-associated mplus expression has quadratic performance, and solves this by using continuation-passing style. Note that there is no problem with bind for a non-determinism monad: like concatMap for lists, it traverses both the left argument and (the result of) the right argument. Kiselyov et al. show how the monad transformer implementation of Hinze can be adapted such that it is also an instance of MonadLogic. Although it can be really tricky to see this directly from the code, this instance of MonadLogic has severe performance problems. Effectively, their implementation of msplit corresponds to converting a difference list to a list and converting to tail of the list to a difference list again. Hence, each invocation of msplit will add one extra operation per result in the remainder of the logical computation.

Their implementation uses continuation-passing style with two continuations, but the point of this chapter is that it is better to make the sequence explicit instead of representing it as a tree of functions (i.e. CPS). Hence, we do not apply our method to this implementation, but to a standard stream implementation of backtracking [Wand and Vaillancourt, 2004] as shown in Figure 5.5(a). In this implementation, the ML type is essentially a list where each node of the list is the result of a computation in the underlying monad. The list can be empty (Nothing) or a head and tail (Just (a,ML m a)). The definitions are then analogous to the definitions for the lists: mplus is concatenation and ≫= is like concatMap.

Notice that ML is *not* the same as the ListT construction:

```
newtype ListT m a = ListT { runListT :: m [a] }
instance Monad m ⇒ Monad (ListT m) where ...
```

This construction only yields a monad if the argument monad, m, is commutative [Jones and Duponcheel, 1993]. The difference is that in ML each node in the "list" is the result of a computation in the underlying monad, whereas with the ListT construction the *entire* list is the result of a single computation in the underlying monad.

An example of the asymptotic performance problem is the following function which obtains at most n solutions of a logical computation.

```
seqN :: MonadLogic m ⇒ Int → m a → m [a]
seqN n m
  | n ≡ 0       = return []
  | otherwise   = msplit m ≫= λx → case x of
                    Nothing    → return []
                    Just (a,m) → liftM (a:) (seqN (n−1) m)
```

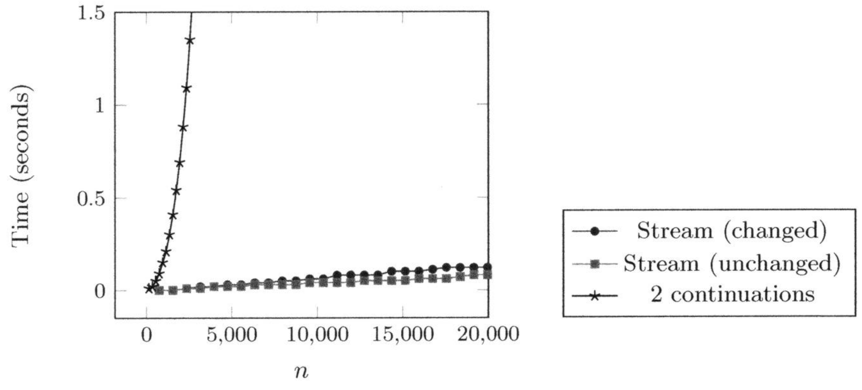
```
newtype ML m a = ML { toView :: m (Maybe (a, ML m a)) }
fromView = ML
single a = return (Just (a,mzero))

instance Monad m ⇒ Monad (ML m) where
  return = fromView ∘ single
  (toView → m) >>= f = fromView $ m >>= λx → case x of
      Nothing    → return Nothing
      Just (h,t) → toView (f h `mplus` (t >>= f))
  fail _ = mzero

instance Monad m ⇒ MonadPlus (ML m) where
  mzero = fromView (return Nothing)
  mplus (toView → a) b = fromView $ a >>= λx → case x of
    Nothing    → toView b
    Just (h,t) → return (Just (h,t `mplus` b))

instance MonadTrans ML where
  lift m = fromView (m >>= single)
instance Monad m ⇒ MonadLogic (ML m) where
  msplit (toView → m) = lift m
```

(a) Original implementation.

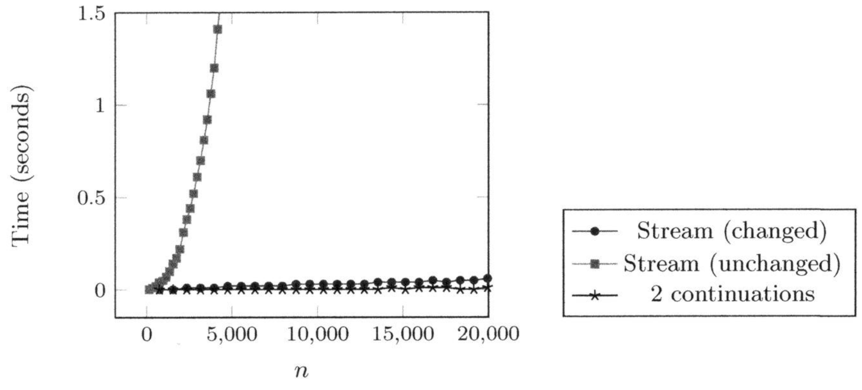
```
newtype ML m a = ML ( CQueue (m (Maybe (a, ML m a))))
fromView = ML ∘ singleton

instance Monad m ⇒ MonadPlus (ML m) where
  mzero = ML empty
  mplus (ML a) (ML b) = ML (a ⋈ b)

toView :: Monad m ⇒ ML m a → m (Maybe (a, ML m a))
toView (ML s) = case viewl s of
  EmptyL → return Nothing
  h ◁ t → h >>= λx → case x of
    Nothing      → toView (ML t)
    Just (hi,ML ti) → return (Just (hi,ML $ ti ⋈ t))

-- the other code is unchanged
```

(b) Changes to the original implementation.

Figure 5.5: A stream implementation of MonadLogic



(a) Running time splitting a logical computation of natural numbers n times.



(b) Running time of observing all results in a left-associated mplus expression with n elements.

Figure 5.6: Running time of msplit and mplus micro benchmarks for LogicT.

Figure 5.6(a)[7]shows, for different implementations, the running time of obtaining n natural numbers using seqN, where the natural numbers are defined as follows[8]:

```
nats = natsFrom 1 where
  natsFrom n = return n `mplus` natsFrom (n + 1)
```

Obtaining a number of solutions requires us to recursively split the logical computation, and hence the two continuation implementation as implemented in Hackage package LogicT has quadratic running time. Of course, this is just a micro-benchmark constructed to illustrate the problem. However, this problem does not only occur on the natural numbers: it occurs *any time* we request only some, instead of all, solutions to a logical computation. This is highly counter-intuitive: it is much faster to obtain *all* results than some results. Moreover, since we are talking about monad *transformers*, requesting all results is not always an option: it may invoke undesired and/or irrevocable effects in the underlying monad.

The same problem occurs with the interleave operator as described by Kiselyov et al., which ensures fair consideration between two branches of a logical computation. An example usage of this operator is the following the logical computation:

```
unfair = do x ← nats `mplus` return 0
            if x ≡ 0 then return x else mzero
```

The behavior of mplus in these implementations is that it first considers all solutions from its left argument, and only afterwards considers the solutions of its right argument. Since nats has an infinite number of results, this computation will never yield a solution. If interleave is used instead of mplus, then solutions from nats and return 0 are considered alternately and the computation will yield a solution. This interleave operator is defined in terms of mplus and msplit as follows:

```
interleave :: m a → m a → m a
interleave l r = msplit l ≫= λx → case x of
   Nothing    → r
   Just (h,t) → return h `mplus` interleave r t
```

Since interleave recursively splits the remaining computation of both arguments, any usage of it while using a two continuation implementation of backtracking will lead to performance problems. For instance, the following logical computation:

```
test = choose [1...n] `interleave` choose [n...1]
   where choose l = foldr mplus mzero (map return l)
```

also runs in $O(n^2)$. The same problem occurs when using using the fair conjunction operator, which is defined in terms of interleave. The cut and soft cut operators are also problematic, but much less severely: they only split the logical computation once.

Obtaining only a limited number of solutions and using the interleaving or fair conjunction operators is not problematic when using the ML implementation of MonadLogic:

[7]These measurements are the median of 5 runs and were performed on an AMD Phenom II X4 905e Processor CPU running Linux 3.2.0 on binaries produced with the GHC 7.6.3 (optimization level 2). The fixed stream implementation uses a worst case constant time catenable queue.

[8](a `f` b) is an alternative notation for (f a b).

```
data It i a = Get (i → It i a) | Done a

instance Monad (It i) where
  return = Done
  (Ret x) ≫= g = g x
  (Get f) ≫= g = Get (f ⋙ g)

get :: It i i
get = Get return
```

Figure 5.7: Iteratees before applying our solution.

we can observe results directly by running a computation in the underlying monad: there is no conversion involved. Instead, the problem is now mplus: it traverses the left hand argument but not the right hand argument. Figure 5.6(b) shows the running time of obtaining all solutions of a left-associated mplus expression:

```
test :: MonadPlus m ⇒ Int → m Int
test n = foldl mplus mzero (map return [1...n])
```

Now the running time of the ML implementation is quadratic. The dual continuation implementation does not suffer the same problem, as it was originally derived by Hinze to solve this problem. Hence, that the performance characteristics of the ML implementation are opposite to those the two continuation implementation: the ML implementation has quadratic performance on a left-associated mplus expression, but no performance problem with msplit.

Applying our solution to the ML implementation yields the changes that are shown in Figure 5.5(b). The changes are very similar to the changes to the (non-generic) Tree data type: we change the ML data type to an explicit expression involving mplus, and the actual logic of non-deterministic choice is moved to the toView function. As can be seen from the graphs, after applying our method the problem with mplus disappears: the running time is now linear. Moreover, this stream implementation with our method applied to it is the *only* implementation which efficiently supports *both* msplit *and* mplus.

5.6.2 Iteratees and related monads

As a second example of how we can apply our solution to a practical example, consider iteratees [Kiselyov, 2012]: a style of incremental input processing that overcomes the problems of lazy I/O and handle-based I/O. We consider a simplified version of iteratees where an iteratee is a monadic computation that can request an input element, as shown in Figure 5.7.

An iteratee is in one of two possible states: the constructors of the It data type. If an iteratee is Done it simply carries the value it produces. If an iteratee needs an input element, it is a Get value, carrying a function that when given the input element returns the next iteratee state. A Monad instance for such iteratees is then defined straightforwardly. In this definition, the (⋙) operator is Kleisli composition (f ⋙ g = λx → f x ≫= g) as introduced in section 5.2.3.

Although it can be easy to miss, the definition of the monadic bind, like its definition in the original paper, exhibits the problematic pattern: it traverses its left argument but not its right argument. It does not matter that (≫=) invokes itself by using function composition instead of application, this just obfuscates the problem.

As example of the performance problem is the following iteratee computation, that gets n elements from the input and then returns their sum:

```
sumInput :: Int → It Int Int
sumInput n = Get ( foldl (⋙) return ( replicate (n − 1) f))
  where f x = get ≫= return ∘ (+ x)
```

Where replicate n e is a function that creates a list of the length n, where each element is e. The sumInput function yields an expression of the form:

```
Get (((( return ⋙ f) ⋙ f) ⋙ f) ... ⋙ f)
```

Figure 5.8 shows that when the argument to Get is called with a new input element x, it costs $O(n)$ steps to obtain the next iteratee state:

```
Get ((((( return ∘ (+ x)) ⋙ f) ⋙ f) ⋙ f) ... ⋙ f)
```

This very similar to the original expression, exhibiting the same problem. Hence, the running time of feeding this iteratee computation n elements and obtaining their sum is quadratic. The sumInput function can easily be made to run in linear time by simply switching from foldl to foldr. However, in general solving such performance problems by avoiding the problematic pattern is not as simple: we must then make sure that that each left argument to bind cannot be the result of a bind.

We can solve the problem with repeated binds by using the codensity monad transformer, as defined in Section 5.3.2, as proposed by Voigtländer [Voigtländer, 2008]. When using this method, we only use codensity transformed iteratees to build monadic expressions:

```
type ItCo i a = CodensityT (It i) a
```

We then redefine get so that it gives a codensity transformed iteratee:

```
getCo :: ItCo i i
getCo = rep get
```

A monadic expression built in this way will then always result in a right-associated expression when converted to a regular iteratee computation, thus avoiding the problem of repeated binds.

We now find ourselves in a familiar situation: this method makes alternating between building and observing problematic. An example of this is the following, often useful, parallel iteratee composition function, defined as a regular (non-codensity transformed) iteratee function:

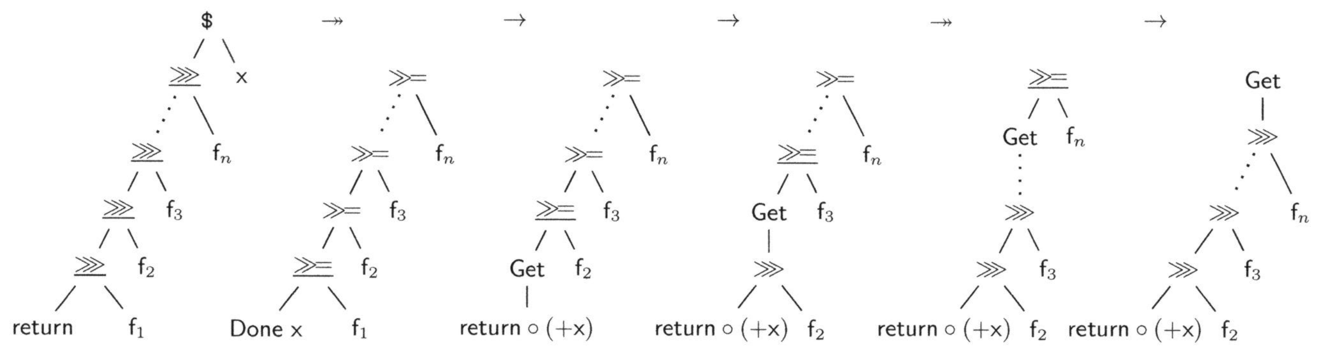


Figure 5.8: Example of an inefficient iteratee computation. The subscript i in f_i indicates the index of the occurrence of f.

```
par :: It i a → It i b → It i (It i a, It i b)
par l r
  | Done _ ← l = Done (l,r)
  | Done _ ← r = Done (l,r)
  | Get f ← l, Get g ← r = get ≫= λx → par (f x) (g x)
```

This operator runs both iteratees in parallel, feeding each input element to both, until at one of the iteratees is done. Afterwards, the remaining iteratee computation of both arguments is returned, which can then be composed again with other iteratees using par and ≫=. The par function is an instance of monadic reflection: we observe the internal state of both iteratees.

If we want to use par on codensity transformed iteratees, we need to redefine it as follows:

```
parCo :: ItCo i a → ItCo i b
        → ItCo i (ItCo i a, ItCo i b)
parCo l r = rep (par (abs l) (abs r)) ≫=
              (λ(l,r) → return (rep l, rep r))
```

We need to eliminate the codensity transformer using abs to observe the states of both iteratees. After applying the original par function, we want to be able to compose the resulting iteratees again with ≫= and parCo. However, they are no longer codensity transformed iteratees, while other iteratees are in this form to avoid the problems with bind. We need to convert the rest of the resulting iteratees back to codensity transformed form. Hence, each invocation of parCo adds an extra operator per Get in the remaining iteratee, which can easily lead to performance problems when iteratees are long lived and used in many invocations of parCo.

A related construction is monadic *coroutines*, which are like iteratees except that they also output an element each time they request an input element. Blažević [Blažević, 2011] presents an extensive library for such coroutines, but his coroutine definition suffers from the same problem as the original iteratee definition.

Another guise of the same situation occurs in monadic FRP (Chapter 4): a framework which essentially applies coroutines in a functional reactive programming (FRP) setting. In monadic FRP, a combinator very similar to par is at the heart of composing reactive computations and the bind in that chapter has the same problem as the original iteratees. In fact, the motivation for this work is that we noticed that our monadic FRP program became progressively slower, due to repeated application of bind on the results of par, and eventually came to a grinding halt. Since par is used often in monadic FRP, and coroutines can live for a long time, being used in many invocations of par, the use of the codensity monad would also lead to a severe slowdown. With our solution applied, monadic FRP programs no longer become progressively slower, running efficiently no matter what the usage pattern.

Our solution can be applied to iteratees, coroutines and monadic FRP. By using an efficient type aligned sequence data structure, the performance of improves dramatically, without constraining ourselves by disallowing functions involving monadic reflection like par. We do not show the code for this due to space considerations, but instead note that iteratees, coroutines and monadic FRP are all instances of a construction known as a *free monad*, which we discuss and show the improved code of in the next section.

5.6.3 Free Monads

Swierstra [Swierstra, 2008] shows how a monad instance can be defined for any functor, resulting in a monad that is called the *free monad* [Awodey, 2006] on that functor. This construction is defined as follows:

```
data FreeMonad f a = Pure a
                   | Impure (f (FreeMonad f a))

instance Functor f ⇒ Monad (FreeMonad f) where
  return = Pure
  (Pure x)   ≫= f = f x
  (Impure t) ≫= f = Impure (fmap (≫= f) t)
```

Swierstra then notes that several well known monads are free monads. For example, the Maybe monad is the free monad on the following functor:

```
data One a = One deriving Functor
```

Now (Pure a) corresponds to (Just a) and (Impure One) corresponds to Nothing.

However, for many functors this construction leads to asymptotic problems. Consider for example the following Functor:

```
newtype Get i a = Get (i → a) deriving Functor
```

A free monad on this functor corresponds to the iteratees we saw in the previous section. Free monads over the following functors:

```
data Node a = Node a a deriving Functor
data Yield out inn a = Yield out (inn → a) deriving Functor
```

correspond to the generic trees with substitution and coroutines, respectively. It should come as no surprise that the performance problem of iteratees, generic trees and coroutines did not go away by formulating them as free monads. Again, we could use continuation-passing style, but this would make functions like par expensive.

We solve these problem *for all* free monads by simply applying our solution. The definition of free monads then becomes:

```
type FC f a b = a → FreeMonad f b
type FMExp f a b = TCQueue (FC f) a b

data FreeMonad f a where
  FM :: FreeMonadView f x → FMExp f x a → FreeMonad f a
data FreeMonadView f a = Pure a
                       | Impure (f (FreeMonad f a))
fromView x = FM x tempty
```

```
toView :: Functor f ⇒ FreeMonad f a → FreeMonadView f a
toView (FM h t) = case h of
   Pure x →
    case tviewl t of
       TEmptyL → Pure x
       hc ◁ tc → toView (hc x ≫̂= tc)
   Impure f → Impure (fmap (≫̂= t) f) where
 (≫̂=) :: FreeMonad f a → FMExp f a b → FreeMonad f b
 (FM h t) ≫̂= r = FM h (t ⋈ r)

instance Monad (FreeMonad f) where
  return = fromView ∘ Pure
  (FM m r) ≫= f = FM m (r ⋈ singleton f)
```

Notice that this code is very similar to the code we got from applying our solution to our generic tree example in Section 4.2. This should come as no surprise: generic trees are free monads.

As usual, the code for these adapted free monads is included in the code accompanying this chapter, as well as a benchmark demonstrating the performance problem and that our method solves it.

5.6.4 Extensible effects

Recently Kiselyov, Sabry, Swords and Foppa introduced *extensible effects* [Kiselyov et al., 2013]: a framework for composing and implementing computational effects that overcomes the problems of monad transformers in terms of efficiency, expressiveness and ease of notation. In this framework an effect is an interaction between a client and a handler: the client sends a value describing the desired effect to the handler, which in turn executes the desired effect and passes the result to the client.

The approach of Kiselyov et al. uses functors to describe both which effect to request and how to continue afterwards. For example, both the request to modify a state and how to proceed afterwards, are represented by the following functor:

```
data ModifyState s w =
  ModState (s → s) (s → w) deriving Functor
```

The first argument tells the handler how to modify the state, whereas the second argument tells the handler how to continue afterwards, it takes the new state and then produces some w. The free monad over this functor is then the value that is interpreted by the handler: if the value is Impure (ModState f c), it applies the function f to the state and calls the function c with the new state. This may again yield an Impure value and the process continues until the handler sees a Pure value.

The *extensible* in extensible effects comes from the fact that handlers do not interpret a free monad over a single functor, but a free monad over an *open union* of functors. An open union is a value that can be of any type in a *set* of types. This distinguishes it from a closed union, for example Either a b, which has a *list* of types. Kiselyov et al. then show an implementation of an open unions of functors, which in itself is again a functor. In this way handlers for different effects can be stacked: if a handler does not handle the desired effect, the value describing the effect is passed to the next handler in the stack.

However, as we saw in the previous section, many functors give rise to performance problems when using a (non-adapted) free monad. For functors describing effects, this is the case if the effect produces some result which is then passed to a continuation function. This is always the case, except for exceptions.

Kiselyov et al. avoid this problem by using a variant of free monads using continuation-passing style. This has the advantage that it avoids the performance problems of wrong groupings of expressions involving bind, but it has the disadvantage that handlers must be written in continuation-passing style. In a related paper, Kammar et al. [Kammar et al., 2013] avoid the performance problem by (implicitly) applying the codensity monad.

Both approaches lead to performance problems when effects requiring reflection such as iteratees, LogicT transformers or delimited continuations are modeled. With our solution, extensible effects can directly be expressed as (adapted) free monads over open unions, without the need for manual continuation-passing style or the codensity monad. Moreover, effects that require reflection *can* then be efficiently supported. An example implementation of extensible effects as efficient free monads is included in the code accompanying this chapter, as well as a benchmark involving reflection in the form of a logical cut effect, that is quadratic in the original implementation, but linear in our adapted implementation.

5.7 Conclusion

Associative operators that traverse their left argument, but not their right argument, can lead to asymptotic overhead. A popular cure is to use continuation-passing style, but this cure is only effective if our usage is strictly separated into a build and an observation phase, otherwise the cure is as bad as the disease.

We presented a solution that solves such performance problems for any usage pattern, even when alternating between building and observing. Our solution reveals a hidden sequence, namely repeated applications of such a problematic operator, and makes it concrete using an efficient sequence data structure.

To support operators where the type of the right argument depends on the type of the left argument, such as the monadic bind, we introduced a generalization of sequences called type aligned sequences. Type aligned sequences enforce the ordering of their elements, and hence rule out ordering bugs.

Monadic reflection, i.e. a way to observe, or reify, the internal state of a monadic computation requires us to alternate between building and observing. We showed that reflection does not have to lead to remorse: our solution efficiently supports reflection. We have demonstrated that our solution can yield an asymptotic running time improvement in iteratees (and related constructs), LogicT transformers, free monads and extensible effects.

Our solution is not limited to the examples we discussed in this chapter. In the accompanying code, we show how sequences can be factored out in delimited continuations [Dyvbig et al., 2007] and term monads [Lin, 2006]. Given the simplicity of the problematic pattern and the widespread usage of continuation-passing style, we suspect that there are many more applications of our solution hiding in corners where we have not looked yet.

Acknowledgment

We thank Jan Rutten, Koen Claessen and the anonymous reviewers for helpful discussions and comments on this chapter.

6

Conclusions

Ideally, programs are described by composing generic, reusable parts. However, sometimes such a composition comes at a cost: the resulting program is too slow, uses to much memory or does not give high enough precision. In this thesis, we made this situation less likely in the domain of interactive visualizations and in the introduction we posed a series of research questions that aim at achieving this goal. In this chapter we discuss the answers to each of these questions, how our results make abstractions more efficient and their limitations.

Graphics abstractions

Research Question 1 *Is it possible to program 2D graphics in a declarative way that is general, simple, expressive, composable and resolution-independent while still being efficient?*

In Chapter 2, we have presented a library for declarative resolution-independent 2D graphics called *Deform* in Chapter 2. Our library generalizes and simplifies the functionality of traditional frameworks, while preserving their efficiency. Our approach is more general than traditional approaches, since we allow the description of arbitrary shapes, arbitrary textures and arbitrary transformations, while traditionally only Bézier curves, a limited set of textures and affine transformations are supported. We have shown that an implementation of a focus+context lenses in our framework gives better image quality and better performance than a solution using a traditional framework, at a fraction of the code.

When using a traditional graphics framework discretizations can be scaled up and sampled to produce other discretizations. In these frameworks it is not possible to analyze a representation of a drawing and hence the only way to produce non-affine transformations is to sample discretizations of a drawing. Apart from producing results with low image quality, this is also very inefficient. In Deform, we delay discretization till the very last moment, costing less time and leading to higher image quality for shapes that are not Bézier curves, textures that are not gradients or images and non-affine transformations.

A shortcoming of our approach is that we currently do not support post-processing image filters such as blurs. Such post-processing filters are typically described by how they combine samples from the original image. Further research is needed to see if such filters can also be described in a resolution independent way while still being efficient. It might be possible to describe such filters using integrals, and then distill an efficient sampling implementation from the description and its usage.

Our approach enables combining advanced effects in interactive visualizations. For example, it becomes much easier to add a focus+context lens to a Google maps-like application. It also enables the description and combination of procedurally generated textures [Ebert, 2003], while maintaining high image quality. An interesting application is computer-generated art. The cover shows an example of computer generated art using Deform. Such art can also be generated using the Context Free tool[1], and their website has many fascinating examples. Support for non-affine transformations has already been discussed on the Context Free forums as a desirable feature, and they could be supported using the insights from Chapter 3. Non-affine transformations open up many new possibilities for computer generated art.

Layout abstractions

Research Question 2 *Is there a linear time algorithm for producing non-layered layouts of trees?*

In Chapter 3, we have shown that there is a linear time algorithm for producing non-layered layouts of trees. This is the fastest one could hope for, since all the nodes in a tree need to be visited at least once to assign them a position. The algorithm we presented is a modification of the Reingold-Tilford algorithm. Since the the original complexity proof of that algorithm uses an invariant that does not hold for the non-layered case, we have presented an alternative complexity proof of the algorithm and its extension to non-layered drawings. Existing extensions to the Reingold-Tilford algorithm which make the layout more visually appealing also work in our algorithm, but they then cause a quadratic running time complexity. We have presented a modification to this extension and proved that this restores the linear running time. This adds an efficient layout algorithm to the toolbox of interactive visualization programmers[2].

Event abstractions

Research Question 3 *How can we efficiently support incremental evaluation in FRP?*

[1]`http://contextfreeart.org`

[2]See `http://www.treevis.net` for an overview of tree visualizations, including our work.

In Chapter 4, we have presented Monadic FRP, a novel formulation of Functional Reactive Programming. We have shown how incremental evaluation in Monadic FRP can be supported simply and in a purely functional way without introducing glitches: inconsistencies due to an incorrect order of updates. This purely functional, incremental update model makes time-branching efficient: we can efficiently roll back and replay reactive computations. We also showed how we can efficiently support time in this system, such that the reactive program is only updated when needed, whereas other FRP-systems require polling.

Our FRP framework currently has the disadvantage that the results of computations over time cannot be shared using the standard `let` construct and that it is not clear how to define mutually recursive reactive computations, such as two sliders in a temperature conversion application that influence each other.

A remaining problem with FRP in purely functional languages is that the outside world must be hooked up to the FRP program with non-FRP code. This is non-modular: each new input/output must be manually routed trough the FRP program and hooked up. The input/output code and the FRP code are highly coupled, and hence it is not desirable to separate these two. We are currently working on an efficient FRP framework, inspired by Monadic FRP, that solves this problem and does not have the disadvantages of Monadic FRP.

Creating interactive programs is tough. An engineer from Abobe stated that 1/3 of the code of Abobe's desktop products (such as Photoshop, Reader and Flash) is event handling code, but that 1/2 of the bugs are in this code [Parent, 2006]. Monadic FRP is a step towards making programming interactive systems much easier and more composable, while still being efficient. Since FRP is a deterministic model, it may also be easier to prove the absence of bugs than in traditional event handling systems.

Research Question 4 *Can series of associative operators be made efficient no matter what the association pattern for all usage patterns?*

In Chapter 5, we have presented a technique that makes the performance of any series of associative operator independent of the association pattern (i.e., the placement of brackets) and the usage pattern, i.e. even when we alternate between constructing the series and pattern matching on its result. Our technique uses efficient catenable sequence datastructures, which solve the dependence on the association pattern for sequence concatenation for any usage pattern. For some (nearly) associative operators, such as the monadic bind, the type of the right argument depends on the type of the left argument. To be able to apply our solution in a type-safe way in such situations, we introduced type-aligned sequences. These type aligned sequences are a generalization of ordinary sequences and allow us to store, for example, sequences of functions, where the output type of each function is the input type of the next function. We demonstrated that our solution solves previously undocumented, severe performance problems in Monadic FRP (Chapter 4), iteratees [Kiselyov, 2012], LogicT transformers [Kiselyov et al., 2005], free monads [Swierstra, 2008] and extensible effects [Kiselyov et al., 2013].

While this work was motivated from a specific problem in Monadic FRP, the problem we solve is far more general. Our results make efficient sequence datastructures applicable to a wider class of problems. Not only does it make them applicable to associative operators, but type-aligned sequences allow us to describe any sequence-like structure efficiently, such as categories, monads and co-monads. An earlier paper [Greif, 2011] lists several use-cases for type-aligned lists, namely parser combinators and type-safe descriptions of syntax and staged interpreters. We expect that efficient type aligned sequences may be used to improve performance in these areas as well.

To conclude, we have shown that abstractions do not have to come at high costs: it *is* possible to create interactive visualizations by composing them from simple abstractions, without paying in terms of performance.

Bibliography

Awodey, S. (2006). *Category theory*. Oxford University Press.

Bergstra, J. A. and Klop, J. W. (1985). Algebra of communicating processes with abstraction. *Theoretical computer science*, 37:77–121.

Blažević, M. (2011). Coroutine pipelines. *The Monad Reader*, 19:29–50.

Bloesch, A. (1993). Aesthetic layout of generalized trees. *Software: Practice and Experience*, 23:817–827.

Borning, A., Marriott, K., Stuckey, P., and Xiao, Y. (1997). Solving linear arithmetic constraints for user interface applications. In *Proceedings of the 10th annual ACM symposium on User interface software and technology*, pages 87–96. ACM.

Brookes, S. D., Hoare, C. A. R., and Roscoe, A. W. (1984). A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599.

Buchheim, C., Jünger, M., and Leipert, S. (2006). Drawing rooted trees in linear time. *Software: Practice and Experience*, 36(6):651–665.

Carpendale, M. S. T. and Montagnese, C. (2001). A framework for unifying presentation space. In *Proceedings of the 14th annual ACM Symposium on User interface software and technology*, UIST '01, pages 61–70, New York, NY, USA. ACM.

Casciola, G. and Morigi, S. (1996). Reparametrization of NURBS curves. *International Journal of Shape Modeling*, 2:103–116.

Claessen, K. (2004). Functional pearl: Parallel parsing processes. *Journal of Functional Programming*, 14:741–757.

Cooper, G. H. and Krishnamurthi, S. (2006). Embedding dynamic dataflow in a call-by-value language. In *Proceedings of the '06 European Symposium on Programming*, pages 294–308.

Courtney, A. (2001). Frappé: Functional reactive programming in Java. In *Proceedings of the '01 International Symposium of Pratical Aspects of Declarative Languages.*

Courtney, A. and Elliott, C. (2001). Genuinely functional user interfaces. In *Proceedings of the '01 Haskell Workshop.*

Courtney, A., Nilsson, H., and Peterson, J. (2003). The Yampa arcade. In *Proceedings of the '03 Haskell Workshop*, pages 7–18.

Dokken, T. and Thomassen, J. (2003). Overview of approximate implicitization. *Topics in Algebraic Geometry and Geometric modelling, AMS series on Contemporary Mathematics CONM 334*, 28(1):169–184.

Dyvbig, R. K., Peyton Jones, S., and Sabry, A. (2007). A monadic framework for delimited continuations. *Journal of Functional Programming*, 17(6):687–730.

Ebert, D. S. (2003). *Texturing & modeling: a procedural approach.* Morgan Kaufmann.

Elliott, C. (1998). Functional implementations of continuous modeled animation. In *Proceedings of the 10th International Symposium on Principles of Declarative Programming*, pages 284–299.

Elliott, C. (2001). Functional image synthesis. In *Proceedings of Bridges.*

Elliott, C. (2004). Programming graphics processors functionally. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, Haskell '04, pages 45–56, New York, NY, USA. ACM.

Elliott, C. (2009a). Beautiful differentiation. In *International Conference on Functional Programming (ICFP).*

Elliott, C. (2009b). Push-pull functional reactive programming. In *Proceedings of the '09 Haskell Symposium.*

Elliott, C. and Hudak, P. (1997). Functional reactive animation. In *Proceedings of the 1997 International Conference on Functional Programming*, ICFP '1997, pages 163–173.

Filinski, A. (1994). Representing monads. In *Proceedings of the 21th Symposium on Principles of Programming Languages*, pages 446–457.

Finne, S. and Jones, S. P. (1995). Pictures: A simple structured graphics model. In *In Glasgow Functional Programming Workshop, Ullapool.*

Friedman, D. P. (1988). The mystery of the tower revealed: A nonreflective description of the reflective tower. *LISP and Symbolic Computation*, 1(1):298–307.

Gibbons, J. (1996). Deriving tidy drawings of trees. *Journal of Functional Programming*, 6(3):535–562.

Greif, G. (2011). Thrists: Dominoes of data. Internet draft. `http://omega.googlecode.com/files/Thrist-draft-2011-11-20.pdf`.

Hasan, M., Rahman, M. S., and Nishizeki, T. (2003). A linear algorithm for compact box-drawings of trees. *Networks*, 42(3):160–164.

Hinze, R. (2000). Deriving backtracking monad transformers. In *Proceedings of the 5th International Conference on Functional Programming*, pages 186–197.

Hinze, R. and Paterson, R. (2006). Finger trees: A simple general-purpose data structure. *Journal of Functional Programming*, 16(2):197–217.

Hoffmann, C. M. (1993). Implicit curves and surfaces in CAGD. *IEEE Computer Graphics and Applications*, 13:79–88.

Hudak, P., Courtney, A., Nilsson, H., and Peterson, J. (2003). Arrows, robots, and functional reactive programming. In *'02 Summer School on Advanced Functional Programming*, pages 159–187.

Hughes, J. (1986). A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22(3):141 – 144.

Hughes, J. (2000). Generalising monads to arrows. *Science of Computer Programming*, 37(1-3):67–111.

Jaskelioff, M. (2009). Modular monad transformers. In *Transactions on Programming Languages and Systems*, pages 64–79.

Jeffrey, A. S. A. (2013). Causality for free!: Parametricity implies causality for functional reactive programs. In *Programming Languages meets Program Verification*, PLPV '13.

Johnson, B. and Shneiderman, B. (1991). Tree-maps: a space-filling approach to the visualization of hierarchical information structures. In *Proceedings of the '91 IEEE Conference on Visualization*, pages 284 –291.

Jones, M. P. and Duponcheel, L. (1993). Composing monads. Research Report YALEU/DCS/RR-1004, Yale University.

Kammar, O., Lindley, S., and Oury, N. (2013). Handlers in action. In *Proceedings of the '13 International Conference on Functional Programming.*

Kaplan, H. and Tarjan, R. E. (1999a). Purely functional, real-time deques with catenation. *Journal of the ACM*, 46(5):577–603.

Kaplan, H. and Tarjan, R. E. (1999b). Purely functional, real-time deques with catenation. *Journal of the ACM*, 46(5):577–603.

Karczmarczuk, J. (1999). Geometric modelling in functional style. In *Proceedings of the III Latino-American Workshop on Functional Programming, CLAPF'99*, pages 8–9.

Karczmarczuk, J. (2002). Functional approach to texture generation. In *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*, PADL '02, pages 225–242, London, UK, UK. Springer-Verlag.

Katifori, A., Halatsis, C., Lepouras, G., Vassilakis, C., and Giannopoulou, E. (2007). Ontology visualization methods—a survey. *ACM Computing Surveys (CSUR)*, 39(4):10.

Kennedy, A. (1996). Drawing trees. *Journal of Functional Programming*, 6(3):527–534.

Kiselyov, O. (2012). Iteratees. In *Proceedings of the 11th International Symposium on Functional and Logic Programming*, pages 166–181.

Kiselyov, O., Sabry, A., and Swords, C. (2013). Extensible effects: An alternative to monad transformers. In *Proceedings of the '13 Symposium on Haskell*, pages 59–70.

Kiselyov, O., Shan, C., Friedman, D. P., and Sabry, A. (2005). Backtracking, interleaving, and terminating monad transformers (functional pearl). In *Proceedings of the 10th International Conference on Functional Programming*, pages 192–203.

Kleiberg, E., van de Wetering, H., and Van Wijk, J. J. (2001). Botanical visualization of huge hierarchies. In *Proceedings of the IEEE Symposium on Information Visualization 2001*, pages 87–94.

Klint, P., Lisser, B., and van der Ploeg, A. (2011). Towards a one-stop-shop for analysis, transformation and visualization of software. In *Proceedings of the Fourth International Conference on Software Language Engineering (SLE 2011)*, pages 1–18. Springer.

Knuth, D. E. (1971). Optimum binary search trees. *Acta Informatica*, 1:14–25.

Lamping, J., Rao, R., and Pirolli, P. (1995). A focus+context technique based on hyperbolic geometry for visualizing large hierarchies. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 401–408. ACM.

Lanza, M. and Ducasse, S. (2003a). Polymetric views - a lightweight visual approach to reverse engineering. *Software Engineering, IEEE Transactions on*, 29(9).

Lanza, M. and Ducasse, S. (2003b). Polymetric views - a lightweight visual approach to reverse engineering. *Transactions on Software Engineering*, 29:782–795.

Liang, S., Hudak, P., and Jones, M. (1995). Monad transformers and modular interpreters. In *Proceedings of the 22nd Symposium on Principles of Programming Languages*, pages 333–343.

Lin, C. (2006). Programming monads operationally with unimo. In *Proceedings of the 11th International Conference on Functional Programming*, pages 274–285.

Ma, Y. L. and Hewitt, W. T. (2003). Point inversion and projection for NURBS curves and surfaces: control polygon approach. *Comput. Aided Geom. Des.*, 20(2):79–99.

Maier, I. and Odersky, M. (2012). Deprecating the Observer Pattern with Scala.react. Technical report, LAMP.

Marriott, K. and Sbarski, P. (2007). Compact layout of layered trees. In *Proceedings of the 13th Australasian conference on Computer science - Volume 62*, ACSC '07, pages 7–14. Australian Computer Society, Inc.

Marriott, K., Sbarski, P., van Gelder, T., Prager, D., and Bulka, A. (2011). Hi-trees and their layout. *IEEE Transactions on Visualization and Computer Graphics*, 17(3):290–304.

Matlage, K. and Gill, A. (2009). ChalkBoard: Mapping functions to polygons. In *Proceedings of the Symposium on Implementation and Application of Functional Languages*.

Mcbride, C. and Paterson, R. (2008). Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13.

Meyerovich, L. A., Guha, A., Baskin, J., Cooper, G. H., Greenberg, M., Bromfield, A., and Krishnamurthi, S. (2009). Flapjax: a programming language for Ajax applications. In *Proceedings of the '09 Conference on Object oriented programming systems languages and applications*, pages 1–20.

Miyadera, Y., Anzai, K., Unno, H., and Yaku, T. (1998). Depth-first layout algorithm for trees. *Information Processing Letters*, 66:187–194.

Moen, S. (1990). Drawing dynamic trees. *IEEE Software*, 7:21–28.

Nguyen, Q. V. and Huang, M. L. (2002). A space-optimized tree visualization. In *Information Visualization, 2002. INFOVIS 2002. IEEE Symposium on*, pages 85–92. IEEE.

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

Nilsson, H., Courtney, A., and Peterson, J. (2002). Functional reactive programming, continued. In *Proceedings of the '02 Haskell Workshop*, pages 51–64.

Okasaki, C. (1995). Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5:583–592.

Okasaki, C. (1998). *Purely Functional Data Structures*. Cambridge University Press.

Parent, S. (2006). A possible future of software development. Presentation. `http://stlab.adobe.com/wiki/images/0/0c/Possible\_future.pdf`.

Paterson, R. (2001). A new notation for arrows. In *Proceedings of the '01 international conference on Functional programming*, pages 229–240.

Peterson, J., Hager, G., and Hudak, P. (1999a). A language for declarative robotic programming. In *Proceedings of the 1999 International Conference on Robotics and Automation.*

Peterson, J., Hudak, P., and Elliott, C. (1999b). Lambda in Motion: Controlling robots with Haskell. In *Proceedings of the 1th International Workshop on Practical Aspects of' Declarative Languages*, PADL 1999.

Peterson, J., Hudak, P., Reid, A., and Hager, G. (2001). FVision: A declarative language for visual tracking. In *Proceedings of the '01 International Workshop on Practical Aspects of Declarative Languages*, PADL '01, pages 304–321.

Pietriga, E. (2005). A Toolkit for Addressing HCI Issues in Visual Language Environments. *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 145–152.

Pietriga, E., Bau, O., and Appert, C. (2010). Representation-independent in-place magnification with Sigma lenses. *IEEE Transactions on Visualization and Computer Graphics*, 16:455–467.

Ragan-Kelley, J., Adams, A., Paris, S., Levoy, M., Amarasinghe, S., and Durand, F. (2012). Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics*, 31(4):32.

Reingold, E. M. and Tilford, J. S. (1981). Tidier drawings of trees. *IEEE Transactions on Software Engineering*, 7(2):223–228.

Robertson, G. G., Mackinlay, J. D., and Card, S. K. (1991). Cone trees: animated 3d visualizations of hierarchical information. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 189–194. ACM.

Schieber, B. and Vishkin, U. (1988). On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal on Computing*, 17(6):1253–1262.

Schneider, P. J. (1990). An algorithm for automatically fitting digitized curves. In Glassner, A. S., editor, *Graphics gems*, pages 612–626. Academic Press Professional, Inc., San Diego, CA, USA.

Sculthorpe, N. and Nilsson, H. (2009). Safe functional reactive programming through dependent types. In *Proceedings of the '09 International conference on Functional programming*, ICFP '09, pages 23–34.

Sederberg, T. W., Anderson, D. C., and Goldman, R. N. (1984). Implicit representation of parametric curves and surfaces. *Computer Vision, Graphics, and Image Processing*, 28(1):72–84.

Stein, B. and Benteler, F. (2007). On the generalized box-drawing of trees: Survey and new technology. In *Proceeding of I-KNOW '07*.

Supowit, K. and Reingold, E. (1983). The complexity of drawing trees nicely. *Acta Informatica*, 18:377–392.

Swierstra, W. (2008). Data types à la carte. *Journal of Functional Programming*, 18(4):423–436.

Tollis, I. G., Di Battista, G., Eades, P., and Tamassia, R. (1998). *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall.

van der Ploeg, A. (2013). Monadic functional reactive programming. In *Proceedings of the 2013 Symposium on Haskell*, pages 117–128.

Van Wijk, J. J. and Van de Wetering, H. (1999). Cushion treemaps: Visualization of hierarchical information. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 73–78. IEEE.

Vaucher, J. G. (1980). Pretty-printing of trees. *Software: Practice and Experience*, 10(7):553–561.

Voigtländer, J. (2008). Asymptotic improvement of computations over free monads. In *Proceedings of the 9th International Conference on Mathematics of Program Construction*, pages 388–403.

Von Landesberger, T., Kuijper, A., Schreck, T., Kohlhammer, J., van Wijk, J. J., Fekete, J.-D., and Fellner, D. W. (2011). Visual analysis of large graphs: State-of-the-art and future research challenges. *Computer graphics forum*, 30(6):1719–1749.

Wadler, P. (1987). Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 307–313, New York, NY, USA. ACM.

Walker, II, J. Q. (1990). A node-positioning algorithm for general trees. *Software: Practice and Experience*, 20:685–705.

Wand, M. and Vaillancourt, D. (2004). Relating models of backtracking. In *Proceedings of the 9th International Conference on Functional Programming*, pages 54–65.

Wetherell, C. and Shannon, A. (1979). Tidy drawings of trees. *IEEE Transactions on Software Engineering*, 5(5):514–520.

Xiaohong, L. and Jingwei, H. (2010). An improved generalized tree layout algorithm. In *Proceedings of the 2nd international Asia conference on Informatics in control, automation and robotics - Volume 2*, CAR'10, pages 163–166. IEEE Press.

Summary

Abstractions, such as functions and methods, are an essential tool for any programmer. Abstractions encapsulate the details of a computation: the programmer only needs to know what the abstraction achieves, not how it achieves it. However, using abstractions can come at a cost: the resulting program may be inefficient. This can lead to programmers not using some abstractions, instead writing the entire functionality from the ground up.

In this thesis, we present several results that make this situation less likely when programming *interactive visualizations*. We present results that make abstractions more efficient in the areas of graphics, layout and events.

Graphics abstractions Graphics abstractions, which for example allow us to draw a line, fill a shape or rotate a drawing, are an essential part of programming interactive visualizations. We present a new declarative approach to resolution-independent 2D graphics that generalizes and simplifies the functionality of traditional frameworks, while preserving their efficiency. Our framework makes it easier to produce high image quality and makes non-affine transformations more efficient. As a real-world example, we show that the implementation of focus+context lenses gives higher image quality and better performance than a previous solution, at a fraction of the code.

Layout abstractions Interactive visualizations often use abstractions that produce some kind of layout, such as a force directed graph layout or a treemap. When laying out trees in a node-link diagram, a classical algorithm exists that produces the layout in linear time, but the resulting layout takes more space than necessary. We present a novel algorithm that also runs in linear time, but produces more compact drawings.

Event abstractions Interactive visualizations need to deal with events such as mouse clicks and touch commands. Dealing with such events is traditionally done with either blocking I/O or callbacks. However, the former requires concurrency to compose reactive parts, which leads to non-determinism. The later leads to inversion of control: the control-flow of the program is dictated by the events that occur, not by the programmer.

An alternative that does not have these problems is Functional Reactive Programming (FRP). However, FRP often comes at the cost of efficiency: parts of the program are re-computed even though nothing changes. We present a novel FRP framework,

called Monadic FRP, that has an efficient, incremental evaluation mechanism, hence preventing such redundant re-computations.

A general problem, which also manifests itself in Monadic FRP, is that for certain associative operators the number of steps it takes to evaluate an expression depends on how the brackets are placed. A solution is to use continuation passing style, but this again imposes a penalty if we alternate between using the associative operator and observing the results of that operator. We present a general solution that makes the performance of such operators efficient regardless of the placement of the brackets, while also providing efficient support for observing the result of that operator.

To conclude, we have shown that abstractions do not have to come at high costs: it *is* possible to create interactive visualizations by composing them from simple abstractions, without paying in terms of performance.

Samenvatting

Een belangrijk instrument voor elke programmeur is het gebruik van *abstracties*, zoals functies en methoden. Abstracties verbergen de details van een berekening, zodat de programmeur alleen maar hoeft te weten wat een abstractie berekent, niet hoe de berekening er in detail uitziet. Het gebruik van abstracties kan echter het ongewenste effect hebben dat het resulterende programma niet efficiënt genoeg is. Om voldoende efficiëntie te bereiken gaan programmeurs sommige abstracties vermijden en in plaats daarvan de gewenste functionaliteit van de grond af aan opbouwen.

Het doel van dit proefschrift is om abstracties voor *interactieve visualisaties* efficiënter te maken. Ik zal me daarbij richten op efficiëntere abstracties voor het maken van twee-dimensionale beelden, layouts van bomen en interactiviteit.

Abstracties voor het maken van twee-dimensionale beelden Abstracties voor het maken van twee-dimensionale beelden, zoals het tekenen van een lijn, het inkleuren van een vorm of het roteren van een tekening, zijn belangrijke gereedschappen bij het programmeren van interactieve visualisaties. Ik presenteer een verzameling nieuwe declaratieve abstracties voor het maken van resolutie-onafhankelijke, twee-dimensionale beelden, die het makkelijker maken om een hoge beeldkwaliteit te bereiken en ervoor zorgen dat niet-affiene transformaties direct uitdrukbaar en efficiënt zijn. Als voorbeeld laat ik zien dat het programmeren van een lokale vergroting, de zogenaamde focus+context lens, met deze nieuwe abstracties een betere beeldkwaliteit geeft, efficiënter is, en minder code vergt dan een implementatie van dezelfde vergroting met bestaande abstracties.

Abstracties voor layouts van bomen Interactieve visualisaties zijn vaak gebaseerd op een layout van een boomstructuur. Voor het berekenen van de layout van bomen bestaat er een klassiek algoritme waarbij het benodigde aantal stappen lineair is in de grootte van de boom. In sommige gevallen produceert dit algoritme echter een layout die meer oppervlakte inneemt dan noodzakelijk. Ik presenteer een aanpassing van dit algoritme die voor een compactere layout zorgt, en bewijs dat het algoritme met deze aanpassing dezelfde tijdscomplexiteit heeft.

Abstracties voor interactiviteit Interactieve visualisaties moeten reageren op gebeurtenissen, zoals het drukken op een muisknop of het ontvangen van een netwerkbericht. De gebeurtenissen worden afgehandeld door òf een methode aan te roepen

die wacht totdat de gevraagde gebeurtenis voorbij is (*blocking I/O*) òf een methode te installeren die wordt aangeroepen als de gebeurtenis optreedt (een *callback*). De eerste methode heeft als nadeel dat het alleen mogelijk is om interactieve programma-onderdelen samen te stellen door middel van parallellisme, wat de complexiteit van het programma aanzienlijk verhoogt. De tweede methode heeft als nadeel dat het leidt tot omkering van de besturing (*inversion of control*): de besturingsstroom (*control flow*) van het programma wordt bepaald door de gebeurtenissen die optreden, in plaats van door de volgorde van de methode-aanroepen die de programmeur heeft aangegeven. Dit maakt het moeilijk om het overzicht te houden.

Een alternatieve aanpak voor het afhandelen van gebeurtenissen die niet leidt tot non-determinisme of omkering van de besturing is Functioneel Reactief Programmeren (FRP). Het gebruik van FRP gaat echter vaak ten koste van de efficiëntie doordat berekeningen onnodig herhaald worden. Ik presenteer een nieuw FRP raamwerk, genaamd Monadic FRP, dat incrementeel te werk gaat, en dus het onnodig herhalen van berekeningen voorkomt.

Een algemeen probleem, dat ook voorkomt in Monadic FRP, is dat voor sommige associatieve operatoren het aantal stappen dat nodig is om de uitkomst van een expressie te berekenen afhangt van van de plaatsing van de haakjes, ook al maakt dit voor de uitkomst niet uit. Een oplossing hiervoor is het gebruik van *continuation passing style*, maar dit is alleen efficiënt als de berekening niet afwisselt tussen het gebruik van de associatieve operator en het observeren van zijn resultaten. Ik presenteer een oplossing die ook in dit geval een snelheidswinst oplevert, zodat het voor alle associatieve operatoren qua efficiëntie niet uitmaakt hoe de haakjes geplaatst zijn.

Het is kortom mogelijk om interactieve visualisaties te maken door ze samen te stellen uit simpele abstracties, zonder dat dit resulteert in een te traag programma.

Titles in the IPA Dissertation Series since 2009

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean.* Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques.* Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time.* Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation.* Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML.* Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange.* Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on* *top of Proof Assistants and making Proof Assistants available over the Web.* Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory.* Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis.* Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks.* Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement.* Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers.* Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics.* Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics.* Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata.* Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software.* Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra.* Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications.* Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10