Expressing Functional Reactive Programming in C++

University of Oulu
Department of Information Processing Science
Master's Thesis
Louai Al-Khanji
11.11.2015

Abstract

Most C++ programs are written in a straight-forward imperative style. While e.g. callbacks are employed either directly or through the observer pattern, the mental overhead of keeping program state congruent is high and increases with program size.

This paper presents a translation of functional reactive programming into C++ terms. This paradigm originates from the Haskell language community and seeks to express easily how programs should react to new input.

Concretely, an implementation of a reactive property class is presented, where property in this context is a class holding a value of a user-specified type. The property class provides a mechanism to bind to it an expression that takes an arbitrary number of inputs, some of which can be other instances of property classes. When any of these dependent properties is updated the expression is re-evaluated, so that a dataflow graph may be built using this type. The automatic re-evaluation reduces the boilerplate code necessary to update variables, which can lead to fewer programming errors and more concise programs.

The implementation demonstrates that the core principles of functional reactive programming can be expressed in modern C++. Further, the implementation can be done in an idiomatic manner which appears familiar to C++ developers.

At the same time, the implementation's complexity highlights how much further the C++ meta-programming facilities must be developed to properly support facilities such as a functional reactive programming library implementation.

A number of compile-time template metaprogramming utilities used in the implementation are also introduced.

Keywords
functional reactive programming, C++, dataflow programming, property classes

Supervisor
Ph.D., University Lecturer, Ari Vesanen

Foreword

When I first picked FRP as the topic of my thesis I thought implementing it in C++ would be relatively straightforward. How naïve I was!

As I continued running into the numerous corner cases of C++, it became clear to me that I knew nothing about the language I had been writing programs in for the past 10 years. Especially the precise semantics of SFINAE and overload resolution were to be discovered only by poring over the wording of the language specification. I learned a lot about C++ to say the least.

Another avenue of research that proved to be educational beyond expectations was the history of reactive and declarative programming. It was rather eye-opening to realize that the literature of the 70's and 80's already covered very large portions of this problem space, a lot of it in what at the time was termed "database research". Perhaps the pendulum will swing back from the stark separation that exists between data and code in many languages today to a joined discussion of the two reminiscent of the homoiconicity encountered in e.g. Lisp.

However that may be, I want to thank all the individuals who helped me produce this thesis. In particular I want to thank my supervisor Ari Vesanen, who very patiently provided me with just the right kind of guidance, encouragement and entertaining chats.

I also want to thank my wife Jonna for her unwavering support while I juggled a full-time job, my university studies and our move to the United States all at the same time. I couldn't have done it without you.

Louai Al-Khanji

San Jose, California
November 11, 2015

Contents

1. Introduction

The purpose of this study is to investigate how reactive programming might be achieved using the C++ programming language, considering specifically the functional reactive programming (FRP) techniques developed within the functional programming community.

The FRP paradigm originates from the Haskell language community and seeks to express easily how programs should react to new input. Conceptually, a dataflow graph is built that captures for each value an expression that can be re-evaluated when any dependent value changes. The concept is called *reactive* because it automatically updates graph nodes when dependent nodes change. A detailed overview of FRP is given in chapter 2.

The main motivating factor is that no ubiquitous implementation for C++ exists at this point. Some large C++ libraries, such as Qt, do employ reactive programming, but do so via custom domain specific languages that are bespoke. Existing research on FRP within non-functional languages is presented in chapter 2.3. Qt in particular is discussed in chapter 4.

Prior research on FRP has focused strongly on functional languages, in particular Haskell, although some research employing other languages does exist. In particular however a literature review did not result in any papers discussing this topic within the context of modern C++ standards ratified after 2011.

The focus of this paper is to develop a working implementation of FRP using plain standard C++, especially using the new language features that are included in C++11 and C++14.

The central research question is the following: *How can the reactive nature of FRP be expressed most naturally in C++, especially in the latest C++ standard versions?* The research method employed is Design Science (Hevner, March, Park, & Ram, 2004).

The main contribution of this paper is a description of a working FRP system in C++. This system is described in detail in chapter 6. Some utility types and routines are also described in Appendices C and D.

Finally, challenges encountered during implementation of this artifact and possible improvements are discussed in chapter 7 and a conclusion is attempted.

2. Concerning Functional Reactive Programming

Functional Reactive Programming (FRP), a type of dataflow programming (Krishnamurthi, 2012, Chapter 17.2.3), is a declarative programming paradigm used to construct reactive systems (Sculthorpe, 2011). In the FRP paradigm, time-varying values are expressed as *behaviors*, and the system is notified using *events* when the value of a behavior has changed (Monsanto, 2009). Implementations of FRP provide constructs to define declaratively the value of one behavior in terms of one or more other behaviors, resulting in a dependency graph through which data updates are propagated (Burchett, Cooper, & Krishnamurthi, 2007).

Although FRP is often tied to purely functional languages (Amsden, 2011), adaptations of the paradigm to imperative languages exist, notably as a Java library (Courtney, 2001) and in C++ as both a language extension adding new grammar (Demetrescu, Finocchi, & Ribichini, 2011) and as a library based on the standard language (Dai, Hager, & Peterson, 2002).

FRP avoids the *inversion of control* typically associated with systems structured around callback functions. A typical pattern associated with inversion of control is backwards structuring of the system – behavior and implementation details of individual system functions tend to bubble up to higher layers of the system in order to be able to respond to external stimuli. By avoiding inversion of control, FRP makes it possible to compose system functionality more readily by enabling e.g. normal nesting of expressions. (Krishnamurthi, 2012, Chapter 17)

2.1 Functional Reactive Programming in Practice

Before undertaking further theoretical discussion of functional reactive programming, it might benefit the reader to gain a more hands-on understanding of the paradigm. We will here present certain examples of applied FRP as presented in the literature.

Czaplicki and Chong (2013) present Elm, an FRP language focusing on the creation of graphical user interfaces (GUIs). Using Elm it is simple to write, for instance, a slide-show:

```
pics = [ "shells.jpg", "car.jpg", "book.jpg" ]
display i = image 475 315 (ith (i `mod` length pics) pics)
count s = foldp (\_ c -> c + 1) 0 s
index = count Mouse.clicks
main = lift display index
```

This short program demonstrates many aspects of functional reactive programming, and for that reason is explained thoroughly. All Elm code presented is adapted from Czaplicki and Chong (2013).

We first define `pics`, a list of pictures that we would like to present one after the other, infinitely. Then we define a function `display` which takes an index and returns an image component showing the corresponding picture. So far there are no surprises.

When we get to the function `count`, however, things become interesting. The function `count` is an example of partial application[1] (Scott, 2009) - it is constructed in terms of another function `foldp`. For any τ and τ′, `foldp` is a function with the signature:

$$(\tau \to \tau' \to \tau') \to \tau' \to signal\ \tau \to signal\ \tau'$$

That is, it takes three arguments, the first being a function with signature $\tau \to \tau' \to \tau'$, the second being a value of type τ' and the third being a value of type $signal\ \tau$. It returns a value of type $signal\ \tau'$. (Czaplicki & Chong, 2013) This is analogous to the regular fold operations encountered in many functional programming languages (Scott, 2009, pp. 545–549). We will cover signals in more detail shortly.

In this case `foldp`'s first, function, argument is constructed using Elm's lambda syntax. The lambda itself takes two arguments, `_` and `c`, corresponding to the first τ and τ′ in the `foldp` signature. The function does not use the first argument, and simply returns `c + 1`.

The second argument to `foldp` is simply the number zero. The third argument is not applied and must be provided to `count`. This is done when `index` is defined, where `Mouse.clicks` is passed as the remaining argument. `Mouse.clicks` is a *signal* provided by Elm. It triggers on every mouse click. (Czaplicki & Chong, 2013)

Finally, `main` is defined. The variable `main` is special in Elm – the value of this variable is displayed on screen. In this case it is defined by applying the function `lift` to `display` and `index`. (Czaplicki & Chong, 2013)

Recall that display is a function taking an integer and returning a `display` – that is, its type is $Int \to display$. `index`, however, is of type $signal\ Int$, and thus cannot be passed to `display` directly. This issue is solved by applying the function `lift`[2], which is of type $(a \to b) \to signal\ a \to signal\ b$ (Czaplicki & Chong, 2013). Knowing this, we can see by simple substitution that the type of `main` is $signal\ display$.

We now return to the subject of signals. Czaplicki and Chong introduce signals as *time-varying values*. In other words, the value of a signal changes over time – conceptually, they are a mapping from time to some value, expressed by Sculthorpe (2011, p. 13) as follows:

$$Time \approx \{t \in R \mid t > 0\}$$

$$Signal\ A \approx Time \to A$$

In Elm, signals change only when certain discrete events happen (Czaplicki & Chong, 2013). In our case above, that event is a mouse button press, which triggers the `Mouse.clicks` signal. Signals are "sticky" – they propagate upwards (Krishnamurthi, 2012), in our case through `count` and `lift` all the way to `main`, exemplifying the dataflow graph mentioned earlier. At each step, a recomputation is triggered as required – first, the

---

[1] Also referred to as *currying* (Scott, 2009)

[2] The function `lift` is actually an application of Monads. Monads are constructs employed in functional programming languages to make it easier to express stateful constructs such as input and output or exceptions (Wadler, 1993). In other word, they are ways to express *side-effects*, "they acknowledge that the physical world is imperative" (Scott, 2009, p. 544).

current `index` is incremented, triggering the generation of a new display, which finally is assigned to `main`. The Elm language runtime reacts in turn to this change to display the new picture.

Note that the program is easy to read and progresses logically[3]. Changing the trigger by which the index is changed is simple. Czaplicki and Chong (2013) give two examples:

```
index2 = count (Time.every (3 * seconds))
index3 = count Keyboard.lastPressed
```

`Keyboard.lastPressed` holds the character that corresponds to the keyboard key that was last pressed (Czaplicki & Chong, 2013). In this case the signal would simply be used as a way to receive wake-up notifications – the actual value of the key is inconsequential.

Here, `index2` would change in intervals of three seconds, and `index3` would change every time a key is pressed on the keyboard. Substituting either for index in the original program would change the behavior accordingly, without modifications to the rest of the program. (Czaplicki & Chong, 2013)

Courtney (2001) also gives an example of a small GUI program in SOE FRP[4]:

```
ball = stretch 0.3 (withColor red circle)
anim = (lift2 move) (p2v mouseB) (constB ball)

main = animate anim
```

We see again a lifting function, in this case with type $(a \to b \to c) \to (Behavior\ a \to Behavior\ b \to Behavior\ c)$. We will cover shortly the difference in terminology, but for a moment consider signals and behaviors as being analogous. To the lifted function `move` are supplied as behavior the current mouse cursor coordinates and the object to move to said coordinate, in this case a red circle of size 0.3. Finally the variable `main` is set by applying the function `animate` to the lifted function. The red circle now tracks the mouse cursor, not a single callback had to be implemented, and inversion of control was entirely avoided.

Now that we have a feel for FRP in practice, let's look at it more holistically.

2.2 Fundamentals of Functional Reactive Programming

According to Sculthorpe (2011, p. 13), "FRP languages can be considered to have two levels to them: a *functional* level and a *reactive* level." Sculthorpe explains that what is meant is that the reactive semantics of FRP are usually embedded in a "purely functional language" acting as a host.

Sculthorpe goes on to describe two types of signals, continuous-time signals and discrete-time signals, forming the reactive part of FRP. Conceptually, continuous-time signals are a mapping from time to some value, as described in the previous section. Discrete-time

---

[3] For a comparison, see Appendix A for a similar program written in C++ using the Qt toolkit

[4] According to Courtney, "So-named because the implementation is described in the textbook "The Haskell School Of Expression". (Courtney, 2001) (Hudak, 2000)

signals on the other hand "are signals whose domain of definition is an at-most-countable set of points in time." That is, they express events which need not have a well-defined mapping to time that can be expressed mathematically. (Sculthorpe, 2011, pp. 13–15)

Further, Sculthorpe explains that there are various branches of FRP and covers two of these, namely Classic FRP (CFRP) and Unary FRP (UFRP). In CFRP, continuous-time signals are referred to as Behaviors, while discrete-time signals are referred to as events. Sculthorpe notes that CFRP's behaviors and events both are actually signal generators in many implementations. Central to CFRP are *lifting functions*, as demonstrated with Elm previously in this paper. (Sculthorpe, 2011, p. 17)

As noted previously, lifting functions are an application of monads. Monads are described well by Wadler (1993) who notes that monads consist of three basic things – a *type constructor*, a *unit* function and a *bind* function[5]. Wadler gives the following signatures for the unit function and bind function respectively: (Wadler, 1993, p. 6)

$$unit :: a \to Ma$$
$$bind :: Ma \to (a \to Mb) \to Mb$$

Here *M* refers to a type constructor for the monad. Wadler describes these two functions clearly:

"First, we need a way to turn a value into the computation that returns that value and does nothing else. ... Second, we need a way to apply a function of type a → M b to a computation of type M a." (Wadler, 1993, p. 6)

Both the function `lift` in Elm and the `lift2` function in SOE FRP presented previously perform a similar task – they take functions that accept regular values into functions that accept reactive values, i.e. signals or behaviors. In other words, using the terminology around monads, they are *unit functions*. Stated more conventionally, they are adapters that take "normal" functions that are not written explicitly for FRP and make these usable in reactive programs – they are the glue that makes FRP easy to use and apply.

Conceptually the entire reactive part of CFRP can be seen as a monad – Sculthorpe's reference to a functional and reactive levels quoted earlier in essence states this. Lifting functions as referred to in the FRP literature essentially perform the same operation as monadic bind functions.

UFRP, as Sculthorpe explains, is explained on the following conceptual model based on signals and signal functions:

$$Signal : Set \to Set$$

$$Signal\ A \approx Time \to A$$

---

[5] Wadler in fact does not name the bind function explicitly in his 1993 paper – in that paper its name is only given as a star: "*A monad is a triple (M; unit; ★) consisting of a type constructor M and two operations of the given polymorphic types.*" (Wadler, 1993, p. 7)

Wadler does in the same paper refer to the function as performing a binding action in multiple places, and we use this naming convention here for clarity.

$$SF : Set \rightarrow Set \rightarrow Set$$

$$SF\ A\ B\ \approx\ Signal\ A \rightarrow Signal\ B$$

Sculthorpe notes that signal functions are the primary entities in UFRP, while signals are second-class citizens that exist only indirectly in said signal functions. UFRP is *unary* indeed because its signal functions take only a single signal and return a single signal. Sculthorpe further notes that discrete-time signals are difficult to express in UFRP, and are usually embedded inside signals as an Event type which conceptually is a temporally ordered and finite list of values. (Sculthorpe, 2011, p. 22)

Sculthorpe also in detail explains *structural dynamism* in the context of FRP. Sculthorpe notes that this is "one of the main things that sets FRP apart from the synchronous data-flow languages". (Sculthorpe, 2011, p. 15) The central idea is that the data graph formed by functions operating on signals is not fixed, and signals can be dynamically replaced by others using *structural switches*. (Sculthorpe, 2011, p. 15) Sculthorpe demonstrates how switching between behaviors can be used to model for instance bouncing balls. (Sculthorpe, 2011, pp. 18–20)

2.3 FRP in Other Languages

As mentioned previously, the core tenets of FRP have been successfully transplanted to a wide variety of different languages from the original implementations in Haskell. A non-exhaustive search through the literature was performed to gain an idea of what kinds of languages have been able to host the reactive level of FRP as identified by Sculthorpe (2011). The following implementations were discovered:

Table 1. FRP Implementations in languages other than Haskell.

Language	**Implementation**
Java	Frappé (Courtney, 2001)
Scheme	FrTime (Cooper, 2008)
Scala	Scala.React (Maier, Rompf, & Odersky, 2010)
Javascript	Flapjax (Meyerovich et al., 2009)
C++	FRP/C++ (Dai et al., 2002)
C++	sfrp (Sankel, 2014)
ML	Adaptive Functional Programming (Acar, Blelloch, & Harper, 2002)

Of interest to a modern C++ implementation is of course the work of Dai et al (2002). Their implementation, termed FRP/C++ by the authors, provides a "baseline" implementation of FRP concepts implemented in C++98. Indeed, their work demonstrates a comprehensive implementation of CFRP, including behaviors, events, switches and the lift operator. The authors note of the implementation that the principle disadvantage lies in the implementation language, specifically the fact that extensive usage of templates and macros results in a "brittle" implementation that can produce hard to understand error messages. (Dai et al., 2002) A principle point of interest then is to attempt to leverage new C++14 language features to produce a more solid implementation.

Examining the FRP/C++ source code and the original paper (Dai et al., 2002) reveals a number of obvious improvements that modern C++ makes possible. In 2002 Dai et al. had to make do without lambdas, without variadic templates and without constexpr, which all visibly affected the implementation. Lambdas are also mentioned in the paper, in which the authors note that they emulate similar functionality with functoids (Dai et al., 2002). These functoids included the number of arguments in the type name, as variadic templates were missing from the language.

A second C++ implementation is Sankel's sfrp library. Presented at the C++Now 2014 conference, this library leverages many modern C++ concepts through the boost libraries. Sankel originally developed this library for an application in robotics, the same domain Dai et al. targeted. Sankel, aware of the prior work by Dai et al, approached the implementation from a C++ perspective and claims thereby to have avoided many of the space and time issues that the work by Dai et al suffered from. (Sankel, 2014)

Of note is the fact that neither of the encountered C++ implementations target the latest C++14 standard. Sankel's sfrp, being more recent, in many ways leverages some of the same concepts through the boost library. However, the library is still constrained in many places to what was possible with C++98, as the implementation was done between 2010 and 2012. (Sankel, 2014) Inspection of the source code reveals that certain C++11 features such as support for rvalue copy construction and rvalue assignment have been retrofitted onto the library, but many of the features now part of the core language are still accessed through the boost libraries.

Also interesting is the Scala.React framework by Maier et al. (2010). The implementation is close in spirit to modern C++ and provides a good comparison of what is possible in other modern programming languages. However, the framework leverages Scala's advanced closures in ways that are impossible in C++, and therefore a simple transliteration of it into C++ is not possible.

Frappé, while interesting, unfortunately loses much of the elegance that FRP provides in its original form. In particular it suffers from overt verbosity due to missing meta-programming facilities in Java – section 4.3 illustrates well how one line of Haskell turns into a dozen, even with some code redacted. (Courtney, 2001, p. 7) We desire a much more succinct syntax.

The other implementations are not comparable to a C++ implementation, owing to their implementation in languages with entirely different memory semantics and programming models in general.

With the above in mind it is believed that an implementation of FRP using modern C++ is novel.

3. Concerning Property Classes

A central tenet of object-oriented programming is message passing. What is meant here is that instead of accessing or mutating directly the data of an object, a message of some form is sent to the object, which "itself selects the *method* by which it will react to the message". (Kim & Lochovsky, 1989, p. 5)

In C++, messages are referred to as member functions, and invoked with the syntax OBJECT.MESSAGE(ARG1,ARG2) (Kim & Lochovsky, 1989, p. 76). The same is true for other languages which model messages as member functions, including for instance C# and Python.

In addition to handling messaging, objects usually also encapsulate data. For instance in the Common List Object System (CLOS) this functionality is exposed in the form of class slots. (Kim & Lochovsky, 1989, p. 52) *Accessor functions* are used to read or write the contents of a slot instead of direct access. (Kim & Lochovsky, 1989, p. 56)

Other languages also support similar data hiding functionality as a core part of the language, often referred to as *properties*.

3.1 Properties in Languages other than C++

Properties are class interfaces that define *accessors* and *mutators*, also referred to as *setters* and *getters*, for a record field of a class. Properties can also include additional functionality, for example change notification delivery or reset functions.

3.1.1 Common Lisp

As mentioned before, Common Lisp supports properties through CLOS slots. Slots are a central aspect of class definitions. Slots have names, accessors and mutators. (Bobrow, Gabriel, & White, 1993)

Bobrow et al give the following usage example: (Bobrow et al., 1993)

```
(defclass algebraic-combination (standard-object)
 ((s1 :initarg :first :accessor first-part)
  (s2 :initarg :second :accessor second-part)))

(defclass symbolic-sum (algebraic-combination)
 ())

(defclass symbolic-product (algebraic-combination)
 ())
```

Here, symbolic-sum and symbolic-product are both subclasses of algebraic-combination, which is a subclass of standard-object. They both have two slots, which can be accessed with the functions first-part and second-part. (Bobrow et al., 1993)

3.1.2 C#

C# includes built-in support for properties. They are specifically designed for data encapsulation using private fields in combination with accessors and mutators, while retaining direct access syntax. (Whitehead, 2008)

Whitehead gives the following example: (Whitehead, 2008)

```
public class GameInfo {
    private string gameName;

    public string Name
    {
        get
        {
            return gameName;
        }
        set
        {
            gameName = value;
        }
    }
}
```

In the code above, the actual string variable gameName is private, while the property Name is public. Elsewhere, the class can be used as follows: (Whitehead, 2008)

```
// Test code
GameInfo g = new GameInfo();

// Call set accessor
g.Name = "Radiant Silvergun";

// Call get accessor
System.Console.Write(g.Name);
```

From the user's point of view, access indeed looks like direct record field access. However, the implementation intercepts the data before it is written to the gameName variable and is free to choose how to react according to the desired application policy and data constraints.

3.1.3 Python

The Python programming language has built-in support for first-class properties. These were introduced in version 2.2 of the language. (Python Software Foundation, 2002)

As of Python 3.4.3, this support takes the form of a class named property that is built into the language. (Python Software Foundation, 2015) The interface is defined as follows:

```
class property(fget=None, fset=None, fdel=None, doc=None)
```

Here, fget and fset are the getter and setter functions respectively, fdel is a deleter function to be used with the del keyword, and doc is a documentation string. (Python Software Foundation, 2015)

As with C#, the property encapsulates data – it does not hold the underlying variable itself. In the release notes for 2.2, this point was specifically raised, as the value might be computed from other data, and not stored directly. (Python Software Foundation, 2002)

In Python, properties can be declared either directly or using decorator mark-up. The Python documentation provides samples of both. (Python Software Foundation, 2015) Direct declaration is as follows:

```
class C:
    def __init__(self):
```

```
        self._x = None

    def getx(self):
        return self._x

    def setx(self, value):
        self._x = value

    def delx(self):
        del self._x

    x = property(getx, setx, delx, "I'm the 'x' property.")
```

Here, the property is declared as a class member explicitly using the property class constructor. The same can be achieved using Python decorators:

```
class C:
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """I'm the 'x' property."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

This code is equivalent to the first example. (Python Software Foundation, 2015)

3.2 Previous C++ Implementations

A number of implementations have been proposed for inclusion in the C++ standard.

In late 2003 Borland proposed an addition to the C++ language that was based on their experience building various GUI systems. The proposal included basic setters and getters for properties in addition to extensions to the C++ run time type information (RTTI) for these objects. (Wiegley, 2002) Borland owns a number of patents in this field (U.S. Patent No. 5,724,589, 1998).

In February 2004 Daveed Vandevoorde summarized the additions to Microsoft's C++/CLI that relate to properties. Though the syntax was slightly different from Borland's earlier proposal, the functionality was largely similar, with a few additions for indexable properties. (Vandevoorde, 2004)

Finally, in April 2004 Lois Goldthwaite proposed an approach that did not rely on any additions to the C++ language. Goldthwaite was not in favor of adding properties to C++, and said so directly: "*A property is a behavior that pretends to be a state. From an OO Design point of view, I think this is A Bad Idea. Behaviour should be visible; state should not.*" Nevertheless, Goldthwaite's proposal did include five property classes implemented using only C++ standard functionality. (Goldthwaite, 2004)

In the end, none of these proposals found their way into the C++ standard.

Outside of the language standard itself, Qt's implementation is a well-known implementation of properties. For example, Borland's paper cited above (Wiegley, 2002) makes reference to Qt's "signal/socket" (sic) implementation, which is directly related to properties in Qt's meta object model (The Qt Company, 2015b).

Qt includes a separate meta object compiler, commonly referred to simply as *moc*. It generates additional meta information about the properties and methods of subclasses of QObject. One part of this meta information includes a list of class properties, declared using a pre-processor macro that the moc can pick up. This includes for example property setters and getters, change notifier signals, and additional meta-information. (The Qt Company, 2015b)

4. Declarative Properties

In addition to abstracting data access using accessor functions and language support for properties, various languages include support for declaratively defining the value of these properties.

The notion of data dependencies has existed for a long time. Even the earliest implementations of Smalltalk, such as Smalltalk-80, included methods to subscribe to object changes. While the mechanism was rudimentary and featured little automation beyond what would today be referred to as the observer pattern, it was already at the time observed that the "*concept of change and update, therefore, are integral to the support of … object dependence relationship.*" (Goldberg & Robson, 1983, pp. 240–243)

When automating these concepts of change and update, the resulting system is often described as being *declarative*. For instance, in the context of database systems, Dayal, Buchmann & McCarthy described their system as follows (emphasis added):

"Active database management systems attempt to provide both modularity and timely response. Situations, actions, and timing requirements are all specified ***declaratively*** *to the system. The system now monitors the situations, triggers the corresponding actions when the situations become true, and schedules tasks to meet both timing requirements and consistency constraints over the shared database, without user or application intervention."* (Dayal, Buchmann, & McCarthy, 1988)

In a similar vein, the Pogo representation system is described by the authors as a "*declarative representation system for graphics*". (Kim & Lochovsky, 1989, p. 155) It is further explained that "*with a declarative representation system, the emphasis is on stating what the desired effect is and not on how the change is to be achieved*". (Kim & Lochovsky, 1989, p. 156)

Clearly such declarative programming concepts are not a new thing. However, the degree to which language support automates such declarative statements has evolved over time.

For instance, to return to the previously mentioned Qt framework, the Qt QML module "*offers a highly readable, declarative, JSON-like syntax with support for imperative JavaScript expressions combined with dynamic property bindings*". (The Qt Company, 2015a)

We will shortly explore QML through practical examples. Following this, we contrast it with the FRP methodologies in an attempt to find commonalities.

4.1 Demonstration of Declarative Properties in QML

As mentioned previously, QML has built-in support for declarative programming. In practice this means that creating data flow dependencies is automated to a high degree and works without explicit mention of these dependencies.

For instance, take the following program:

```
import QtQuick 2.4

Image {
    property var pics: [ "shells.jpg", "car.jpg", "book.jpg" ]
```

```
    property int clicks: 0
    source: pics[clicks % pics.length]

    MouseArea {
        anchors.fill: parent
        onClicked: clicks++
    }
}
```

This program is functionally identical to the Elm gallery sample given in chapter 2. The `source` property of the `Image` item is updated automatically whenever the `clicks` property is incremented, demonstrating reactive functioning. In QML this is referred to as a *binding*.

Similarly, the SOE FRP example demonstrating the ball following the mouse cursor is simple to express:

```
import QtQuick 2.4
import QtQuick.Window 2.2

Window {
    visible: true

    MouseArea {
        id: mouse
        anchors.fill: parent
        hoverEnabled: true
    }

    Rectangle {
        x: mouse.mouseX; y: mouse.mouseY
        width: 30; height: width
        radius: width; color: "red"
    }
}
```

In contrast to the FRP implementations inspected previously it is clear that QML relies extensively on mutable variables. Nevertheless, the reactive aspect is quite similar, and as shown in the examples given above does not usually require using a `lift` function.

While the examples above demonstrate properties that act more like event types in CFRP, the language also natively supports a notion of behaviors. We can slightly modify the mouse-following example above to demonstrate this:

```
import QtQuick 2.4
import QtQuick.Window 2.2

Window {
    visible: true

    MouseArea {
        id: mouse
        anchors.fill: parent
        hoverEnabled: true
    }

    Rectangle {
        x: mouse.mouseX; y: mouse.mouseY
        width: 30; height: width
        radius: width; color: "red"
```

```
        Behavior on x { NumberAnimation {} }
        Behavior on y { NumberAnimation {} }
    }
}
```

The two added lines are marked in bold.

Here, the `x` and `y` properties are defined to have a `Behavior` that holds a `NumberAnimation`. Instead of tracking the mouse cursor position instantaneously, the coordinates are smoothed out over a period of time. By default this is simply a linear animation over 500 milliseconds, but this can be customized extensively or extended through custom types.

QML bindings can also be declared programmatically, for instance to replace existing bindings or when objects are created dynamically. In this case the usage is very close to the `lift` function discussed previously in the context of Elm.

To illustrate the similarity, consider the following slightly more complex program:

```
import QtQuick 2.4
import QtQuick.Window 2.2
import QtQuick.Controls 1.2

Window {
    id: root; visible: true; title: "Rotating Circles"
    color: "black"; width: 400; height: width

    function radians(d) { return d * Math.PI / 180; }
    function cartesian(r, t) { return Qt.point(r * Math.cos(t),
                                               r * Math.sin(t)); }

    Component {
        id: circleComponent
        Rectangle {
            property var polarOffset
            anchors.centerIn: parent; color: "blue"
            anchors.horizontalCenterOffset: polarOffset.x
            anchors.verticalCenterOffset: polarOffset.y
            width: root.width/20; height: width; radius: width
        }
    }

    Rectangle {
        id: rect; anchors.centerIn: parent
        width: root.width / 2; height: width; radius: width / 8
        NumberAnimation on rotation {
            loops: Animation.Infinite; duration: 5000
            from: 0; to: 360
        }
        Text {
            anchors.centerIn: parent
            text: "%1 circles".arg(parent.children.length-1)
        }
        function addCircle() {
            var rNow = rotation;
            var properties = {
                "polarOffset": Qt.binding(function() {
                    return cartesian(2/3*width,
                                     radians(rotation - rNow) * -2);
                })
```

```
            };
            circleComponent.createObject(this, properties);
        }
    }

    Button { text: "Add Circle"; onClicked: rect.addCircle() }
}
```

Here, a central rectangle rotating clockwise is declared. When the button is pressed, a circle rotating the central rectangle in counterclockwise direction is added. The output looks like this:

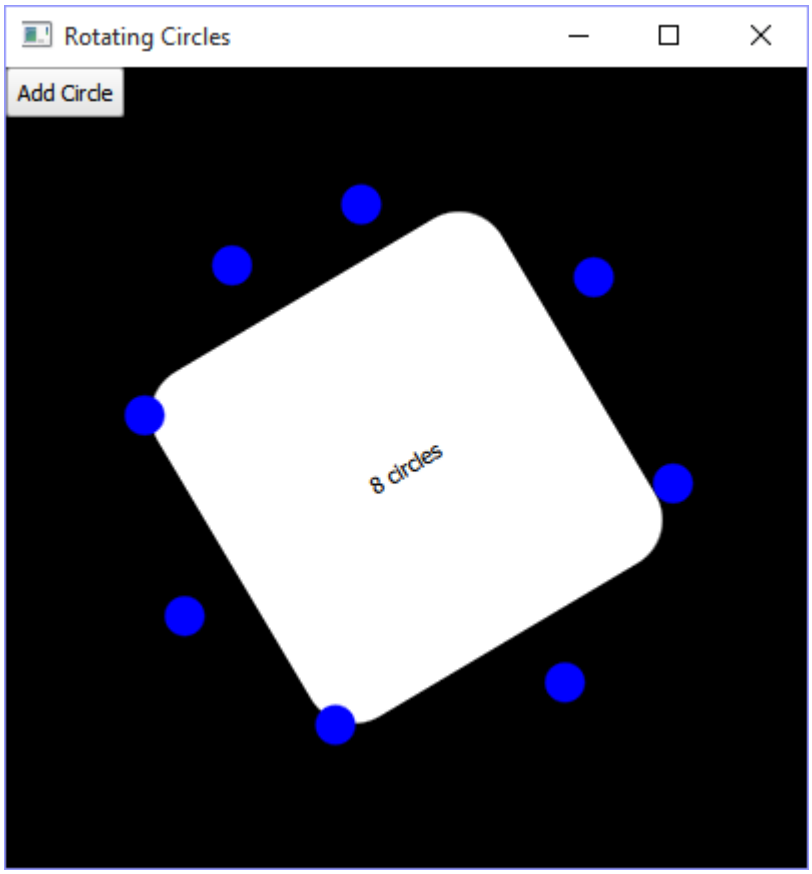

Figure 1. Rotating Circles QML Example

This sample program demonstrates both the advantages and disadvantages of the automated reactive binding in QML. Because the arguments to the lifting function `Qt.binding` are not given explicitly, it can be difficult to pass non-reactive arguments – for this reason, the current rotation must first be copied into the `rNow` variable.

4.2 QML vs. FRP

Declarative properties as employed by QML share many similarities with FRP. The main similarities are as follows:

1. Changes propagate through an automatically generated data graph
2. Functions defined in terms of regular types can be reused easily in the reactive context
3. There are constructs with discrete-time updates
4. There are constructs with continuous-time updates

Interestingly, like FRP with the reactive and functional levels, QML is built in a similar manner. At the bottom lies Qt's meta object system implemented in C++, on which a JavaScript virtual machine is built.

QML and FRP also share the same core constructs. There are FRP's discrete-time signals which are broadly equivalent to properties in QML. Additionally, FRP's continuous-time signals are functionally identical to QML's Behavior type.

QML and FRP differ in the underlying language. Whereas FRP languages are usually based on a functional language, such as Haskell, QML is based on JavaScript. This difference is evident for instance in how the lifting of constructs into the reactive level works. QML attempts to bind dependencies automatically, whereas the FRP languages expect dependencies to be passed explicitly to the lifting function. The latter makes sense when one considers that FRP is often paired with languages that have only immutable variables.

5. Reactive Properties in C++

Based on the information above, the problem can be divided into two parts. First, design and implement an interface for regular properties. Here, the C# implementation can serve as a starting guideline, as the syntax follows C and thus C++ to a large degree. Similarly, Python is not very different.

Layered on top of this basic notion of properties, it is possible to build declarative properties. This is especially valuable and useful with the addition of lambda functions in C++11, but usable even without said functions in earlier versions of the language.

5.1 Problem Definition

Based on the prior work a problem definition can be formulated. A declarative property should:

1. Hold a value of a specific type and provide for its manipulation using e.g. getters and setters
2. Provide a mechanism that binds a reactive expression to the value help by property
3. Respond to update notifications from properties referenced in said expression by sending its own update notification so that a data graph might be built

Ideally the syntax would be similar to the following:

```
declarative_property<int> left_margin(20),
                          content_width(400),
                          right_margin(20);

declarative_property<int> total_width = [](){
    left_margin + content_width + right_margin
};
```

In contrast to the functional languages in which FRP has so far been most widely explored, side effects are an integral part of the C++ programming model. For this reason the property class really is the central point of any implementation of FRP within C++ - where for instance in Haskell one would simply define a new immutable variable through a Monadic detour, in C++ the same is achieved by mutating an existing variable.

Conceptually then, monadic constructs as they exist in the functional languages are expressed in C++ as mutations of the value of variables. Taking into account the monadic nature of the reactive level of FRP as described previously, a translation of the FRP lifting functions into C++ is coupled to the property class, which encapsulates what is essentially a monadic triplet of a type constructor, unit function and bind function – as evidenced by the three items in the list above.

In C++, the property class is the most central building block of FRP.

5.2 First Attempts

First attempts to find a solution were focused on the property class itself. See Appendix B for an example of such an attempt.

It quickly became evident that the interesting part was not in the way the actual data is stored – at the time the monadic parallels were not yet evident, and so the problem was not yet viewed in the light of unit and binding functions.

Instead of attempting to build the perfect property class, it was then decided to focus first and foremost on the reactive level. Subsequently a working solution could be found. This solution is presented below in chapter 6 and the process of design and discovery further discussed in chapter 7.

6. Overview of Designed Artifact

The C++ implementation proved challenging. As type safety was desired much template metaprogramming had to be employed to provide the necessary infrastructure.

We will shortly introduce the public interfaces of the types that have been written. While the public interface may appear simple it masks very complex internal logic. We will dive into the details of the implementations after the public interfaces have been introduced.

6.1 The emitter Class

Most central is the `emitter`. This template class encapsulates what is essentially the observer pattern in a convenient and typesafe manner. It can be employed as follows:

```
rpp::emitter<int, int, std::string> em;
em.connect([] {
    std::cout << "callback 1" << std::endl;
});
em.connect([] (int i) {
    std::cout << "callback 2: " << i << std::endl;
 });
em.connect([] (int i, std::string s) {
    std::cout << "callback 3: " << i << " " << s << std::endl;
});
em.fire(42, "Hello World!");
```

Note that the receiving object need not take all the emitter parameters as arguments. Note also the namespace `rpp` – all the implemented types have been placed in it. However, for brevity we will not always include this namespace in the discussion below.

6.2 The property Class

The `emitter` is used by the `property` class for change notifications. Two `emitter` instances are used, one to signal changes to the value held by the `property` and the other to signal destruction of the `property`. Using the property class is straight-forward:

```
rpp::property<int> c;
c.onChanged.connect([] (int i) {
    std::cout << "c changed: " << i << std::endl;
});
c = 50;
c = 60;
```

The snippet above will print out the following two lines:

```
c changed: 50
c changed: 60
```

Additionally one can connect to `c.aboutToDestruct` for a callback just before the object is destroyed.

The property class also implements bindings, so that we finally have reactive functionality in C++. This can be employed as follows:

```
rpp::property<std::string> first, last, first_and_last;
```

```
first_and_last.onChanged.connect([] (const std::string &s) {
    std::cout << "first_and_last is now: " << s << std::endl;
});

first_and_last.bind([] (const std::string &s1, const std::string &s2) {
    return s1 + " " + s2;
}, first, last);

first = "John";
last = "Doe";

first = "Mike";
first = "Jack";

last = "Jones";
```

The output is the following:

```
first_and_last is now: John
first_and_last is now: John Doe
first_and_last is now: Mike Doe
first_and_last is now: Jack Doe
first_and_last is now: Jack Jones
```

When a dependency is destroyed the binding is also destroyed.

6.3 Implementation Details – emitter

The emitter class is implemented in two parts. Because C++ templates instantiations result in distinct types given distinct template arguments some basic functionality has been separated into a non-template class `emitter_base`.

The main task of this base class is to maintain a list of receivers which are connected to the emitter. This is done with a simple `vector` member:

```
std::vector<std::shared_ptr<receiver_base>> m_connections;
```

The `receiver` class is tied tightly to the `emitter`. `emitter_base` contains a nested definition of `receiver_base`. The `emitter` class template in turn contains a template subclass of `receiver_base`. For clarity, the implementation structure of these classes is as follows:

```
class emitter_base {
protected:
    class receiver_base {
        // implementation omitted
    };
};

template <class... EmissionArgs>
class emitter : public detail::emitter_base
{
    template <class Callable>
    struct receiver : receiver_base {
        // implementation omitted
    };
};
```

The conceptual architecture is also given in figure 2 below.

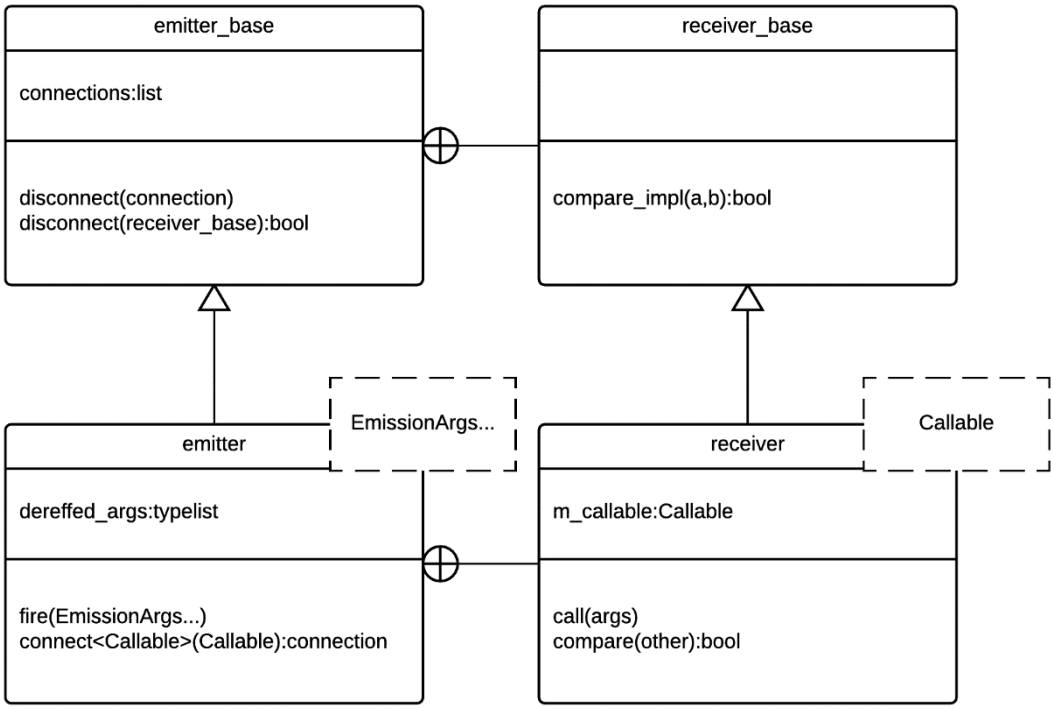

Figure 2. UML class diagram of emitter conceptual architecture

Some explanation is required to clarify this. Here `emitter_base` and `receiver_base` should be clear. However, the `emitter` class template and the nested `receiver` class template are tricky.

The emitter takes as its template arguments the types that it will emit. These are given by the user directly, for instance:

```
rpp::emitter<int, float, std::string> e;
```

Here the parameter pack `EmissionArgs` contains the three given types as arguments. However, the nested receiver template has so far not been instantiated. This is instead done in the connect member function:

```
template <class Callback>
connection connect(Callback cb)
{
    using receiver_t = receiver<Callback>;
    std::shared_ptr<receiver_t> receiver;

    auto cb_ptr = reinterpret_cast<const void *>(std::addressof(cb));
    for (const auto &r : m_connections) {
        if (r->compare(cb_ptr)) {
            receiver = std::static_pointer_cast<receiver_t>(r);
            break;
        }
    }

    if (!receiver) {
        receiver = make_shared<receiver_t>(std::forward<Callback>(cb));
        m_connections.push_back(receiver);
```

```
    }

    return connection(this, receiver);
}
```

This code is non-trivial in many ways. Let us walk through it step by step.

The function returns a connection object which can be used at later stage to break the connection and to query whether the connection is still valid.

On the first line of the function body, the template alias declaration of `receiver_t` actually performs a lot of work. To understand this, we will look at the `receiver` template class in a moment.

The function goes on to get the address of the given callback and attempts to find an existing receiver. If no such receiver exists one is constructed. Finally a connection object referencing the connection is returned to the caller.

6.3.1 Preparing to receive

The `receiver_base` and `receiver` template classes look as follows:

```
class receiver_base
{
public:
    virtual ~receiver_base() {}
    virtual void call(void **args) = 0;
    virtual bool compare(const void *arg) const = 0;

    template <class T,
              class = decltype(std::declval<T>() == std::declval<T>())>
    static bool compare_impl(const void *left, const void *right)
    {
        return *reinterpret_cast<const T *>(left)
                == *reinterpret_cast<const T *>(right);
    }

    template <class>
    static bool compare_impl(...) { return false; }
};

template <class Callable>
struct receiver : receiver_base
{
    Callable m_callable;
    static const int n_accepted = n_args_accepted<Callable,
                                                  dereffed_args>();

    receiver(Callable c)
        : m_callable(std::forward<Callable>(c))
    {
        static_assert(n_accepted >= 0, "");
    }

    void call(void **args) override
    {
        emit_to(
            m_callable,
            args,
            std::make_index_sequence<n_accepted> {},
            utils::typelist_left_t<n_accepted, dereffed_args> {}
```

```
        );
    }

    bool compare(const void *arg) const override
    {
        return compare_impl<Callable>(
          arg,
          reinterpret_cast<const void *>(std::addressof(m_callable))
        );
    }
};
```

Here once again more work is being performed than is apparent at first. Let us however quickly consider the receiver template class without going deeper.

6.3.2 The receiver_base Class

The `receiver_base` class is in most regards simple. As required by C++ for safe destruction of polymorphic classes, the destructor is marked virtual. Additionally two pure virtual member functions are declared, `call` and `compare`. These are implemented by the `receiver` subclass in a type-specialized manner and will be discussed shortly.

The overloaded `compare_impl` template function warrants further explanation. What we have here is an application of the C++ concept SFINAE – Substitution Failure Is Not An Error. This concept is widely employed in C++ template metaprogramming, and is fundamentally quite simple.

When considering overloaded function templates, the C++ compiler might have several potential candidates, what is referred to in the C++ standard as the *overload set*. The overload set is ordered from most to least specialized function. The compiler will iterate through this set and pick the first function where it can successfully substitute the template arguments with the given types. However, invalid functions resulting from this substitution are not a compilation error – the candidate is simply discarded. Hence, the term SFINAE.

In this case two candidates are provided. The first candidate takes two template arguments, but a default argument is given for the second:

```
class = decltype(std::declval<T>() == std::declval<T>())
```

This attempts to deduce the resulting type of an expression comparing two variables of type T. The type deduction can fail – there might not be a defined comparison operator for instance. In this case there is no type to deduce and the expression is ill-formed, which will cause the compiler to remove this function from the overload set that it is currently processing.

In this case the next function in the set will be considered:

```
template <class>
static bool compare_impl(...) { return false; }
```

In C++ the precedence of operator `...` is the lowest of all. We use it here to ensure that this overload will always be the last one to be considered.

These two overloads together provide a way to compare all those types which define a comparison operator, without requiring said operator. In this context we want to be able

to skip double connections to free-standing functions for instance, which can be compared. On the other hand `std::function` is an example of a callable type which can act as a receiver which does not have a comparison operator. By employing SFINAE here we can support the comparable types without failing when a non-comparable type is given as the receiver.

Apart from the `compare_impl` function overloads the `receiver_base` class is straight-forward.

6.3.3 The receiver Cass

The code for the receiver class starts by declaring a variable `m_callable` that contains the given callable object. As this is a template class this can be an object of any type – the only requirement is that it must be copyable. This variable is initialized in the constructor.

The constructor also contains a `static_assert` which checks at compile time that the `n_accepted` member is greater than or equal to zero.

To understand what is happening we must walk through the employed templates one by one. However, let us first look at the other parts of the receiver template class and return here in a moment.

After defining the constructor the `call` and `compare` virtual functions are implemented. The `compare` function is simple and is implemented in terms of the two `compare_impl` functions we discussed previously. The function `call` is also short, and delegates its work to several template constructs, chiefly `emit_to`. We see in both the constructor and in `call` again a reference to `n_accepted`, so let us return to it.

6.3.4 Finding the Right Number of Arguments

Recall that it is possible to connect to an emitter a receiver that takes fewer arguments than the emitter. This is useful for instance if we have a function that must be called when some value changes, but which is not concerned with the new value itself. For this reason it must be known how many of the emitter's parameters to pass to the receiver. The number of arguments to pass is stored in the `n_accepted` variable.

The definition of `n_accepted` refers to `dereffed_args`, which is defined in the `emitter`:

```
using dereffed_args=utils::typelist<std::add_const_t<EmissionArgs>...>;
```

Starting from the inside out, the `std::add_const_t` template is applied to every member of the `EmissionArgs` parameter pack. This is achieved by using the `...` operator to expand the parameter pack.

Because C++ does not provide a native list type at compile-time[6], a helper type `rpp::utils::typelist` was implemented. The internals of this type are not immediately

[6] C++ does include several sequential data structure for use at *runtime*, such as std::list and std::vector for instance. However, we need here a data type which can be used *during compilation*. Here different restrictions apply. See for instance Alexandrescu (2001) for a thorough introduction.

relevant here but are discussed in Appendix C. utils::typelist for compile-time lists. The important thing is that all the arguments to be emitted are marked `const` so that e.g. reference types may not be modified by receivers. The modified types are stored for later use in the alias type `dereffed_args`.

The callable type and the list of argument types are passed to the `n_args_accepted` template. This template returns the number of arguments that the given callable type accepts or `-1` if it cannot be called. For instance, given the function `void f(int, float)` and the types `int`, `float`, `char`, it will return `2`. However, given `void g(std::string)` instead it will return `-1`, since `int` cannot be converted to `std::string` and the function `g` cannot be called with zero arguments[7]. The implementation is as follows:

```
template <class F, class... Args>
struct n_args_accepted
    : std::conditional_t
        < CONCEPT_P(concepts::Callable(F, Args...))
        , utils::int_constant<sizeof...(Args)>
        , n_args_accepted
            < F, utils::left_n_t < sizeof...(Args)-1
                                 , Args...>>>
{};

template <class F>
struct n_args_accepted<F>
    : std::conditional_t
        < CONCEPT_P(concepts::Callable(F))
        , utils::int_constant<  0 >
        , utils::int_constant< -1 >>
{};

template <class F, class... Args>
struct n_args_accepted<F, utils::typelist<Args...>>
    : n_args_accepted<F, Args...>
{};
```

The three templates work together to find the number of arguments that can be passed. The last template is the simplest – it is simply a specialization[8] of the first that takes an `utils::typelist`, unwraps the argument pack that it holds and recurses. This way the `n_args_accepted` template can be transparently used with either native C++ parameter packs or with type lists.

C++11 added std::tuple which can be used at compile time, but unfortunately this class is easy to misuse in a way that adds run-time overhead, since it is designed to contain data.

In contrast, rpp::utils::typelist has no data members – it exists purely to be able to modify lists of template arguments, with the specific design goal of being able to store and manipulate lists of types within the constraints of the C++ template mechanism. The design approach that was taken is similar to lists in Lisp – refer to Appendix C for implementation details.

[7] In contrast with `void g(std::string s = "default")` the connection could be made.

[8] Please see Section 6.3.5 for a discussion of class template specialization

`n_args_accepted` gives its result via `utils::int_constant`[9]. Both the first and the second template inherit from the result of `std::conditional_t`. This template takes three arguments, where the first one is of type `bool`. If this argument is `true` the template evaluates to the second template argument – if it is false, to the third. Both of these templates are standardized versions of constructs first developed by Andrei Alexandrescu. (2001)

The primary `n_args_accepted` template is used when there are potential arguments. If the `CONCEPT_P` macro evaluates to true then the current number of arguments is returned. Otherwise, one argument is removed from the list by a call to `utils::left_n_t`[10] and a recursive call is made. We will discuss the `CONCEPT_P` macro below in a moment.

The second specialization is chosen when there are no arguments to consider, either because the emitter was declared with an empty list of arguments or because all the arguments were discarded by the primary template. This specialization is not recursive and will therefore always terminate the recursion.

6.3.5 Expressing Concepts

We can now see that `n_args_accepted` relies on the macro `CONCEPT_P` to determine whether the callable `F` can in fact be called with the arguments under consideration. `CONCEPT_P` is a convenience macro around an ad-hoc implementation of the C++ feature of Concepts. The relevant code is defined as follows:

```
namespace concepts {
struct Callable {
    template <class F, class... Args>
    auto requires_(F &&f, Args&&... args)
    -> decltype( f(std::forward<Args>(args)...) );
};

template <class Concept, class Enable = void>
struct models : std::false_type {};

template <class Concept, class... Ts>
struct models < Concept(Ts...)
              , utils::void_t<decltype(
        std::declval<Concept>().requires_(std::declval<Ts>()...))>>
  : std::true_type {};

} // namespace concepts

#define CONCEPT_P(...) concepts::models<__VA_ARGS__>::value
```

In this case, the macro `CONCEPT_P` is expanded into

```
concepts::models<concepts::Callable(F, Args...))>::value
```

---

[9] `utils::int_constant` is simply a specialization of `std::integral_constant` and defined as follows:

```
template <int N>
using int_constant = std::integral_constant<int, N>;
```

[10] Refer to Appendix C for a detailed discussion of the `typelist` templates

and similarly for the case with no arguments. Now the compiler must make a choice between the two templates that are available for `concepts::models`, and it does so based on the template arguments that are passed to it. (ISO/IEC, 2014a, para. 14.5.5.2)

The first template class definition, which derives from `std::false_type`, is in C++ terminology the *primary* template, as none of its parameters are specialized. It is the generic fallback definition. For our purposes we desire that it be considered after our second definition. We achieve this by making the second template a *specialization* of the first, in which case the C++ standard requires that the compiler must consider this specialized definition first. (ISO/IEC, 2014a, para. 14.5.5.2)

To be exact, in this case the second template is a *partial template specialization*. In contrast to what would be an *explicit template specialization*, also referred to as a *full template specialization*, a *partial template specialization* provides a specialization for a certain class of template arguments. In other words, they also take template arguments which are evaluated, and in the case of class templates are of the general form

`template` <*parameter-list*$_{opt}$>
`class` *identifier* <*argument-list*$_{opt}$> *declaration* (ISO/IEC, 2014a, sec. A12)

The argument list of the specialization must match the parameter list of the primary template. (ISO/IEC, 2014a, para. 14.5.5.1)

In our specialization the parameters and arguments are as follows after the `CONCEPT_P` macro is expanded, where Parameter refers to the primary template and Argument to the template specialization:

Table 2. Arguments of `concepts::models` class template specialization.

Primary Template Parameter	Specialization's Argument
`Concept`	`Concept(Ts...)`

`Enable`	`utils::void_t`
	`<`
	`  decltype`
	`  (`
	`  std::declval<Concept>()`
	`  .requires_(std::declval<Ts>()...)`
	`  )`
	`>`

The syntax of the first argument is somewhat unusual. In this case we simply want the compiler to map the template arguments passed to the template specialization into the template parameters of the specialization. Consider again the result of the `CONCEPT_P` macro expansion above. Even though it takes the form of a function call, no call is ever made – the compiler simply considers the types in the expression and deduces the template specialization's template parameters `Concept` and `Ts` from the template instantiation site. This is the same syntax used e.g. by `std::function`.

By now the compiler has already all the arguments that were given to the template, and so we must provide an argument for the Enable parameter of the original template

ourselves. Here we use the `void_t` type transformation[11] that was developed by Walter Brown. (2014a)

As the sole argument for `void_t` we give the deduced return type of a call to the `requires_` member function of an instance of `Concept`, which in this case is of type `concepts::Callable`. The parameters given to the function are passed through all the way from the `receiver` template. The first element in the parameter pack is the function being evaluated and the rest the arguments being tried.

Now if the function is callable with the given arguments the return type of `concepts::Callable::requires_()` will be deducible and the specialization of `concepts::models` that inherits from `std::true_type` will be chosen. If this is not the case, the primary template that derives from `std::false_type` will be chosen. Even though in this case only the Callable concept has been implemented, it is possible to extend this mechanism easily.

6.3.6 Making the call

At this stage we have determined how many arguments we can pass to the receiver. The `static_if` in the constructor ensures that an invalid receiver type will result in a compilation error, so we know that a valid call is possible.

The `emitter` template class has a method named fire that is defined as follows:

```
void fire(EmissionArgs... args)
{
    void *a[] = {
        const_cast<void *>(reinterpret_cast<const void *>(&args))... ,
        nullptr
    };
    for (auto &c : m_connections) c->call(a);
}
```

This method is rather simple – it takes all the arguments, casts their addresses to `void` pointer type and stores them in an array. An extra `nullptr` is always appended to the array as well. This is for the case where the argument list is empty to avoid an array of size zero, which is not valid. It then calls the virtual `call` method of each `receiver`, passing in the array. Recall that the call function was defined as follows in the `receiver` template:

```
void call(void **args) override {
    emit_to(m_callable, args,
```

---

[11] `void_t` is defined as per Brown's proposal (Brown, 2014a):

```
template<class...>
struct make_void { using type = void; };

template<class... Args>
using void_t = typename make_void<Args...>::type;
```

The concept behind `void_t` is once again SFINAE. If any of the arguments expands into an invalid construct, the entire expression is ill-formed and the compiler will discard the template that it is currently processing from the overload set. The choice of the type `void` is arbitrary, but it must match the primary template. (Brown, 2014b)

```
            std::make_index_sequence<n_accepted> {},
            utils::typelist_left_t<n_accepted, dereffed_args> {});
}
```

If we consider the arguments being passed to `emit_to` one by one, we now know that

- `m_callable` is the actual callback object,
- `args` is an array of `void` pointers to the values being emitted,
- `n_accepted` is the number of these arguments that should be passed to the callback and hence the third parameter is an index sequence containing the indexes `0 ... n_accepted-1`,
- `dereffed_args` is the list of emission arguments made `const` and `typelist_left` is used to select the first `n` elements of a list, therefore the fourth parameter is a type list of length `n_accepted` holding the target value types to call the callback with.

Therefore, `emit_to` must call the given callable with the target of the pointers cast to the types given. Recall that we have already been informed by the compiler that this call can be done, so this is a safe operation. The `emit_to` template function is defined as follows:

```
template <class F, std::size_t... Is, class... Args>
auto emit_to(F &&f, void **args,
             std::index_sequence<Is...>,
             utils::typelist<Args...>)
{
  (void) args;
  return
  f(*reinterpret_cast<utils::select_nth_t<Is,Args...> *>(args[Is])...);
}
```

Although rather short, the function is quite complex. There are three template parameters, respectively:

1. The callback type `F`
2. A non-type parameter pack `Is` of type std::size_t
3. A type parameter pack `Args`

The actual arguments of the function are:

1. The callback `f`
2. The pointers to the arguments `args`
3. An unnamed `std::index_sequence` to pass `Is` implicitly
4. An unnamed utils::typelist that similarly passes `Args`

Within the function, `args` is first cast to `void` to avoid compiler warnings about unused parameters when the list of arguments to pass is empty. Next, the given function is called and its return value returned.

The interesting aspect of this function is in the parameter casting and unpacking. Starting with the outer part of the expression, we dereference the pointer at `args[Is]` after applying a `reinterpret_cast`. The parameter pack expansion at the end applies to the indexes within `Is`, as `Args` is already explicitly unpacked within the expression. In other words, for each index within Is we `reinterpret_cast` the pointer at that index and pass the dereferenced value to the given function `f`.

The parameter pack `Is` is simultaneously unpacked in two places in this expression. Within the `reinterpret_cast` template argument we employ `utils::select_nth_t` to select the correct target type for the cast from the list of argument types, and as the cast's argument we select the matching pointer from the array that was previously built.

6.4 Implementation Details – property

The property class implements the actual functionality that this paper set out to design. Like the emitter and receiver classes, the property class is also made up of a non-template base class on top of which a templated class is built. The base class is simple:

```
struct property_base {
  virtual ~property_base() { aboutToDestruct.fire(); }
  rpp::emitter<> aboutToDestruct;
};
```

Here we provide an emitter which all property classes will have to notify listeners when the object is destructed. The actual property class is quite short as well:

```
template <class T> struct property : detail::property_base
{
  rpp::emitter<T> onChanged;

  property &operator=(const T &t)
  { value = t; onChanged.fire(value); return *this; }

  operator T() const { return value; }

  template <class Func, class... Args,
    class Result = std::result_of_t<Func(Args...)>,
    class = std::enable_if_t<std::is_convertible<Result, T>::value>>
  void bind(Func&& f, Args&&... args)
  {
    auto b = binding<Result>::create(
      std::forward<Func>(f), std::forward<Args>(args)...);

    b->onUpdated.connect([this] (const Result &r) { *this = r; });
    b->connections.emplace_back(aboutToDestruct.connect([b] {
      delete b;
    }));
    b->update();
  }

private:
    T value {};
};
```

As we can see there is an emitter `onChanged` which is fired when the `property`'s value changes and an assignment operator as well as a conversion operator to access said value. The actual value member is default initialized using the uniform initialization syntax introduced in C++11.

The template member function `bind` is complex and where the core logic that finally implements reactive properties is found. Recall that we want to be able to say something along the lines of the following:

```
property<int> left_margin, width, right_margin, total_width;
total_width.bind([](int a, int b, int c, int d){
  return a + b + c + d;
```

```
}, left_margin, width, right_margin, 20);
```

Keeping that in mind, let us inspect the template parameters. The template takes four parameters, where `Func` and `Args` will usually be deduced at the call site. The third parameter `Result` is deduced using the `std::result_of_t` standard template and gives the return value of the given function with the given arguments. Since the `property` class has a user defined conversion operator to the held type, the result's type can be checked directly through a simulated call. The fourth unnamed parameter is simply a sanity check to ensure that the bound function's return type is in fact convertible to the type that the `property` holds.

The function body is relatively simple. A new binding object is created via a static constructor function. A connection is made to the bindings onUpdated emitter to update the property. A connection is also made to the property's aboutToDestruct emitter in order to destruct the binding object at the same time. Otherwise a dangling binding object might attempt to access memory that is no longer valid. Finally, the update method of the binding is called to execute the binding and thereby retrieve the initial value.

6.4.1 The binding class

We now come to the last piece of the reactive puzzle, the binding class. Once again, there is a non-templated base class:

```
struct binding_base {
  virtual ~binding_base()
  { for (auto &c : std::move(connections)) c.disconnect(); }

  std::vector<rpp::detail::emitter_base::connection> connections;
};
```

Here we simply declare a `vector` of emitter connections which are all disconnected when the `binding` is destroyed. Note that there is one non-obvious aspect in the destruction – the vector is moved from, thereby immediately emptying the member variable even before the `for` loop is entered. This is to guard against infinite loops in case of circular connections – if the destructor is re-entered for any reason the loop will not be executed a second time. The actual binding class template is as follows:

```
template <class T> struct binding : detail::binding_base {
private:
  template <class Func, class... Args>
  binding(Func&& f, Args&&... args)
  {
    func = [f=std::move(f), a=std::tuple<Args...>(args...)] () mutable
           -> decltype(f(args...)) {
             return rpp::utils::invoke_tuple(
               std::forward<decltype(f)>(f),
               std::forward<decltype(a)>(a)
             );
           };

    listen(args...);
  }

public:
  template <class... Args> static inline binding<T> *
  create(Args&&... args)
  { return new binding(std::forward<Args>(args)...); }
```

```
  void listen() {}

  template
  < class U, class... Tail,
    class = std::enable_if_t
     <std::is_base_of<detail::property_base, U>::value == false>
  >
  void listen(U &, Tail&&... tail)
  { listen(std::forward<Tail>(tail)...); }

  template <class U, class... Tail>
  void listen(property<U>& head, Tail&&... tail) {
    connections.emplace_back(head.onChanged.connect([this] {
      update();
    }));
    connections.emplace_back(head.aboutToDestruct.connect([this] {
      std::cerr << "Broken binding: dependent property destructed!"
                << std::endl;
      delete this;
    }));
    listen(std::forward<Tail>(tail)...);
  }

  void update() { onUpdated.fire(func()); }

  std::function<T()> func;
  rpp::emitter<T> onUpdated;
};
```

Because it is possible that a property that is a binding depends on is destructed at any time, it was decided to force the bindings to be placed in dynamic memory. Thereby a binding can call operator delete on itself. For this reason the constructor is marked private and the binding is intended to be constructible only via the static create method.

The create method is simple and simply returns a newly allocated binding object to the constructor of which it forwards all of its arguments. The constructor will interpret the first argument to be a callable function and the rest to be arguments to this function, as is the case in the property::bind method.

Within the constructor a lambda is constructed within which the given arguments are captured. The lambda takes no arguments but returns the desired return type. The lambda must be marked `mutable` as the result of the captured function can vary between invocations. The function arguments are captured into a `tuple` object. The `invoke_tuple` utility function is used to call the given object using the tuple.[12]

Finally, for any `property_base` passed as an argument the binding connects to its `onChanged` and `aboutToDestruct` emitters to react accordingly by either emitting an updated value or by deleting itself.

[12] Please refer to Appendix D for the implementation of this function

7. Discussion

The solution presented here fulfills the basic requirements of FRP and demonstrates that this technique can in fact be transplanted to non-functional languages successfully.

It also demonstrates that C++, even the latest ISO standard revisions, makes it very difficult to successfully develop working compile-time domain specific languages (DSL).

7.1 Implications

The solution presented above provides the basic building blocks for reactive programming in C++. While further improvements are warranted for wide-spread production use, it is shown that the basic principle is sound and does translate into the imperative world of C++.

The main advantage of this approach is that the mental burden placed upon the programmer to remember to update all relevant data structures is decreased. A secondary advantage is that the code is often more compact and readable, as the intent is expressed more directly.

The monadic transliteration presented in chapter 5 is interesting in its own right, as there might be a generic monadic pattern that can be developed for C++ as well. While this problem space has seen work by e.g. Sinkovics and Porkoláb (2013) it remains relatively obscure.

7.2 Complexities of the C++ Programming Language

While the templating mechanism is now widely used to develop sophisticated template metaprograms, the syntax used to do so remains cumbersome and the facilities offered by the language primitive. The fact that the `utils::typelist` and `utils::invoke_tuple` utilities had to be developed from scratch attests to this, as does a look at the implementation of `n_args_accepted` or `emit_to`, for instance.

The other challenge that was encountered when implementing the presented artifact pertained to memory handling, where C++ is uniquely challenging. In pure functional language like Haskell the immutability of values ensures that in e.g. a dataflow graph any update must be complete, and that nodes of the graph cannot simply disappear. In other languages garbage collection of one sort or another achieves the same, e.g. Python, Java and C#.

In the spirit of C++, it was desired not to dictate that either `rpp::emitter` or `rpp::property` must be dynamically allocated. It was also desired that destruction of properties that a bound expression depends upon would not terminate the program due to invalid memory access. In the end it was decided that the expression itself must be able to self-destruct – therefore it must be allocated dynamically. It is only for this reason that the `binding` class exists.

The complexity that the above language properties introduce also played a large part in how the syntax of the final classes came to be. Recall that the “ideal syntax” in chapter 5 looked as follows:

```
declarative_property<int> left_margin(20),
```

```
                            content_width(400),
                            right_margin(20);

declarative_property<int> total_width = []{
    left_margin + content_width + right_margin
};
```

In the final implementation, this would be written like this:

```
rpp::property<int> left_margin(20),
                   content_width(400),
                   right_margin(20),
                   total_width;

total_width.bind([] (int left, int width, int right) {
    return left + width + right;
}, left_margin, content_width, right_margin);
```

While the difference is not large, it does affect the readability. Furthermore, it is a direct result from the language constraints described above.

The `bind` method exists for two reasons. First, it was desired to ensure that the binding object that holds the callable and the references to the properties is in fact allocated dynamically. This is required so that the "self-delete" that is employed in `binding::listen` does not in fact illegally terminate the program. Second, and more importantly, we must explicitly pass the parameters that the expression will require.

The issue of the self-delete could be solved by reimplementing operator new for the `binding` class and setting a flag that indicates that the binding was in fact allocated dynamically. However, this was deemed to be outside of the scope of this paper. It is also another indicator of just how complex implementing correct memory handling can be.

The issue of the parameters however is more vexing. Since C++ does not expose the abstract syntax tree at compile time, discovering these parameters automatically at compile time is not possible. This inherently makes the template mechanism weaker than e.g. Lisp macros. While this is not a large issue within the template mechanism itself, since at compile time everything is strongly typed and lazily evaluated, it prevents certain constructs that integrate with run-time behavior of C++.

There are run-time solutions that could be employed to partially mitigate this. For instance properties could add themselves to a list when their value is queried. This list would be cleared before re-evaluating a bound property, so that the contents would be the dependencies of the expression. Implementing this correctly is far from trivial however – the list would have to be globally accessible from any property, and therefore in thread local storage. Further, a single list is in fact not enough – a stack of lists is required, since querying a property might conceivably trigger a different bound expression to be evaluated, at which point a new list would have to be pushed to the stack in order not to lose the previous information. Finally, this technique breaks down whenever an expression has multiple paths of execution – it would only be possible to capture those properties queried within whichever path happens to be executed. This final problem is insurmountable currently – and because of it, there is no guarantee that the dataflow graph can be maintained correctly, leading back to the issue of memory handling whenever a dependent property is destroyed.

7.3 Possible Improvements to the Designed Artifact

The solution presented here is in many ways simplistic. As discussed above, some implementation details stem from the design of C++, but there are further improvements possible within the current C++ language framework.

Chiefly, the way that rpp::property currently holds its stored value and how it is accessed could be greatly enhanced. This could be done using e.g. policy-based design as discussed by Alexandrescu. (2001)

Some of these improvements might be e.g. different access patterns, as was implemented for example in one of the first attempts given in Appendix B.

An improved property class might for instance have the following signature:

```
template
< class T
, template <class...> class AccessPolicy = owning_access_policy
, template <class...> class NotifyPolicy = emitter_notify_policy
>
struct property;
```

This would make it possible to integrate the class with existing access and notification policies that some C++ libraries might provide, for instance the property mechanism provided by the QObject class of the Qt toolkit (The Qt Company, 2015b) or the signaling mechanism provided by the Boost.Signals2 library (Gregor & Hess, 2009).

7.4 Conclusion

The presented solution demonstrates that FRP concepts can be used in C++ programs, although the approach in many ways does differ. It thereby also demonstrates that FRP is not bound to purely functional languages, despite claims to the contrary.

The limitations encountered within C++ itself do on the other hand also demonstrate that this technique is complex to implement within C++, due to the multi-faceted nature of this language. It is to be hoped that as the C++ language continues to evolve, some of these aspects will have solutions in the future.

It is acknowledge that the implementation has room for improvement, as discussed in the previous chapter. However, many of these improvements are of the kind that previous literature does cover, and was simply deemed out of scope for this paper. More fundamental limitations pertain to e.g. performance in terms of CPU cycles and memory, which was not taken into consideration here. Effects on CPU cache locality for instance have not been measured, and further work on many performance aspects is required.

Another interesting area for future work is integration of reactive types with existing frameworks, such as Qt and/or boost or Win32 COM objects for instance.

References

Acar, U. A., Blelloch, G. E., & Harper, R. (2002). Adaptive Functional Programming. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (pp. 247–259). New York, NY, USA: ACM. doi:10.1145/503272.503296

Alexandrescu, A. (2001). *Modern C++ Design: Generic Programming and Design Patterns Applied*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Amsden, E. (2011). A survey of functional reactive programming. *Unpublished*. Retrieved from http://www.cs.rit.edu/~eca7215/frp-independent-study/Survey.pdf

Bobrow, D. G., Gabriel, R. P., & White, J. L. (1993). CLOS in Context: The Shape of the Design Space. In *Object Oriented Programming: The CLOS Perspective* (pp. 29–61). Cambridge, MA, USA: The MIT Press.

Brown, W. E. (2014a). *TransformationTrait Alias void_t* (C++ Standards Committee Paper No. WG21 N3911).

Brown, W. E. (2014b). *Modern Template Metaprogramming: A Compendium, Part II*. Presented at the cppcon2014, Bellevue, WA, United States. Retrieved from https://youtu.be/a0FliKwcwXE

Burchett, K., Cooper, G. H., & Krishnamurthi, S. (2007). Lowering: A static optimization technique for transparent functional reactivity. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation* (pp. 71–80). ACM.

Cooper, G. H. (2008). *Integrating Dataflow Evaluation into a Practical Higher-order Call-by-value Language*. Brown University, Providence, RI, USA.

Courtney, A. (2001). Frappé: Functional reactive programming in Java. In *Practical Aspects of Declarative Languages* (pp. 29–44). Springer.

Czaplicki, E., & Chong, S. (2013). Asynchronous functional reactive programming for GUIs. In *ACM SIGPLAN Notices* (Vol. 48, pp. 411–422). ACM.

Dai, X., Hager, G., & Peterson, J. (2002). Specifying behavior in C++. In *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on* (Vol. 1, pp. 153–160). IEEE.

Dayal, U., Buchmann, A. P., & McCarthy, D. R. (1988). Rules are objects too: A knowledge model for an active, object-oriented database system. In K. R. Dittrich (Ed.), *Advances in Object-Oriented Database Systems* (pp. 129–143). Springer Berlin Heidelberg. Retrieved from http://link.springer.com.pc124152.oulu.fi:8080/chapter/10.1007/3-540-50345-5_9

Demetrescu, C., Finocchi, I., & Ribichini, A. (2011). Reactive Imperative Programming with Dataflow Constraints. In *ACM SIGPLAN Notices* (Vol. 46, pp. 407–426). ACM.

Goldberg, A., & Robson, D. (1983). *Smalltalk-80: The Language and Its Implementation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Goldthwaite, L. (2004). *C++ Properties -- a Library Solution* (C++ Standards Committee Paper No. SC22/WG21/N1615=04-0055).

Gregor, D., & Hess, F. M. (2009). *Chapter 27. Boost.Signals2 - 1.59.0. The Boost C++ Libraries (BoostBook Subset)*. Retrieved October 18, 2015, from http://www.boost.org/doc/libs/1_59_0/doc/html/signals2.html

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design Science in Information Systems Research. *MIS Q.*, *28*(1), 75–105.

Hudak, P. (2000). *The Haskell School of Expression: Learning Functional Programming through Multimedia* (4 edition.). Cambridge, UK ; New York: Cambridge University Press.

ISO/IEC. (2014a). *International Standard ISO/IEC 14882:2014(E) – Information technology - Programming languages - C++* (Fourth Edition.). Geneva, Switzerland: International Organization for Standardization (ISO).

ISO/IEC. (2014b). *Programming Languages - C++ Extensions for Library Fundamentals* (ISO/IEC JTC1 SC22 WG21 N4082). Geneva, Switzerland: International Organization for Standardization (ISO).

Kim, W., & Lochovsky, F. H. (Eds.). (1989). *Object-oriented Concepts, Databases, and Applications*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co.

Krishnamurthi, S. (2012). *Programming Languages: Application and Interpretation* (Second Edition.). Providence, Rhode Island, USA: Brown University. Retrieved from http://cs.brown.edu/courses/cs173/2012/

Maier, I., Rompf, T., & Odersky, M. (2010). Deprecating the Observer Pattern. Retrieved from http://infoscience.epfl.ch/record/148043

Meyerovich, L. A., Guha, A., Baskin, J., Cooper, G. H., Greenberg, M., Bromfield, A., & Krishnamurthi, S. (2009). Flapjax: A Programming Language for Ajax Applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (pp. 1–20). New York, NY, USA: ACM. doi:10.1145/1640089.1640091

Monsanto, C. (2009). Liftless Functional Reactive Programming. Presented at the The 21st International Symposium on Implementation and Application of Functional Languages (IFL), South Orange, NJ, USA.

Python Software Foundation. (2002). *What's New in Python 2.2*. Retrieved March 5, 2015, from https://docs.python.org/3/whatsnew/2.2.html

Python Software Foundation. (2015). *2. Built-in Functions — Python 3.4.3 documentation*. Retrieved March 5, 2015, from https://docs.python.org/3/library/functions.html#property

Sankel, D. (2014). *Functional Reactive Programming - Cleanly Abstracted Interactivity*. C++Now 2014. Retrieved from https://www.youtube.com/watch?v=tyaYLGQSr4g&feature=youtube_gdata_player

Scott, M. L. (2009). *Programming Language Pragmatics, Third Edition* (3 edition.). Amsterdam ; Boston: Morgan Kaufmann.

Sculthorpe, N. (2011). *Towards safe and efficient functional reactive programming* (PhD thesis). University of Nottingham, Nottingham, United Kingdom.

Sinkovics, Á., & Porkoláb, Z. (2013). Implementing monads for C++ template metaprograms. *Science of Computer Programming*, *78*(9), 1600–1621. doi:10.1016/j.scico.2013.01.002

The Qt Company. (2015a). *QML Applications*. *Qt Documentation*. Retrieved March 5, 2015, from http://doc.qt.io/qt-5/qmlapplications.html#what-is-qml

The Qt Company. (2015b). *The Property System*. *Qt Documentation*. Retrieved March 5, 2015, from http://doc.qt.io/qt-5/properties.html

Vandevoorde, D. (2004). *C++/CLI Properties* (C++ Standards Committee Paper No. SC22/WG21/N1600=04-0040). Edison Design Group.

Wadler, P. (1993). Monads for functional programming. In M. Broy (Ed.), *Program Design Calculi* (pp. 233–264). Springer Berlin Heidelberg. Retrieved from http://link.springer.com/chapter/10.1007/978-3-662-02880-3_8

Whitehead, J. (2008). *Introduction to C# - Properties, Arrays, Loops, Lists*. PDF document. Retrieved from https://classes.soe.ucsc.edu/cmps020/Winter08/

Wiegley, J. (2002). *PME: Properties, Methods and Events* (C++ Standards Committee Paper No. SC22/WG21/N1384=02-0042). Borland Software Corp.

Wold, I. (1998). *U.S. Patent No. 5,724,589*. Washington, DC: U.S. Patent and Trademark Office.

Appendix A. Slide Show with C++ and Qt

```
class SlideShow : public QLabel
{
    Q_OBJECT

    int _index {};
    QStringList _images;

public:
    SlideShow(QStringList images) : QLabel(), _images(images) {
        resize(475, 315);
        setScaledContents(true);
        updatePicture();
    }

    void updatePicture() {
        if (!_images.empty()) {
            setPixmap(_images.at(_index));
        }
    }

protected:
    void mousePressEvent(QMouseEvent *) {
        _index = (_index + 1) % _images.size();
        updatePicture();
    }
};

int main(int argc, char **argv)
{
    QApplication app(argc, argv);
    QStringList pics { "shells.jpg", "car.jpg", "book.jpg"  };
    SlideShow slideShow(pics);

    slideShow.show();
    return app.exec();
}
```

Appendix B. First Attempt

```
enum property_access_mode {
    property_readwrite,
    property_readonly,
    property_writeonly
};

template <typename T>
static constexpr auto makeDefaultGetter(T &t) {
    return [&t] () -> T {return t;};
}

template <typename T>
static constexpr auto makeDefaultSetter(T &t) {
    return [&t](const T &other){t = other;};
}

template <typename T, property_access_mode = property_readwrite>
class property;

template <typename T>
class property<T, property_readwrite> {
public:
    using value_type = std::decay_t<T>;

    property() = delete;
    property(const property &other) = delete;
    property &operator=(const property &other) = delete;

    property(value_type &t)
        : property(makeDefaultGetter<T>(t), makeDefaultSetter(t))
    {}

    template <typename getter_t>
    property(getter_t getter, value_type &t)
        : property(getter, makeDefaultSetter(t))
    {}

    template <typename setter_t>
    property(value_type &t, setter_t setter)
        : property(makeDefaultGetter<T>(t), setter)
    {}

    template <typename getter_t, typename setter_t>
    property(getter_t getter, setter_t setter)
        : _getter(getter), _setter(setter)
    {}

    operator T() const {
        return _getter();
    }

    void operator=(const value_type &t) {
        _setter(t);
    }

private:
    std::function<T()> _getter;
    std::function<void(const value_type &)> _setter;
};
```

```
template <typename T>
class property<T, property_readonly> {
public:
    using value_type = std::decay_t<T>;

    property() = delete;
    property(const property &other) = delete;
    property &operator=(const property &other) = delete;

    property(value_type &t) : property(makeDefaultGetter<T>(t)) {}

    template <typename getter_t>
    property(getter_t getter) : _getter(getter)
    {}

    operator T() const {
        return _getter();
    }

private:
    std::function<T()> _getter;
};

template <typename T>
class property<T, property_writeonly> {
public:
    using value_type = std::decay_t<T>;

    property() = delete;
    property(const property &other) = delete;
    property &operator=(const property &other) = delete;

    property(value_type &t) : property(makeDefaultSetter(t)) {}

    template <typename setter_t>
    property(setter_t setter) : _setter(setter) {}

    void operator=(const value_type &t) {
        _setter(t);
    }

private:
    std::function<void(const value_type &)> _setter;
};

class TestClass {
    float _delta{};
    float _alpha{};

    void setAlpha(float a) {
        _delta = a / 2.0;
        _alpha = a;
    }

public:
    TestClass()
        : delta(_delta)
        , alpha(_alpha,
                [this](const float &a){setAlpha(a);})
    {}

    property<float> delta;
    property<float> alpha;
};
```

```
void test() {
    TestClass testClass;

    testClass.delta = 100;
    std::cout << "alpha: " << testClass.alpha << " delta: " <<
testClass.delta << std::endl;

    testClass.alpha = 66;
    std::cout << "alpha: " << testClass.alpha << " delta: " <<
testClass.delta << std::endl;

    testClass.alpha = 200;
    std::cout << "alpha: " << testClass.alpha << " delta: " <<
testClass.delta << std::endl;
}
```

Appendix C. utils::typelist for compile-time lists

The lack of native list types for C++ template metaprograms has been discussed since the technique's discovery. The traditional way of solving these is well presented e.g. by Alexandrescu (2001) and is well established. In essence the structure is the same as in the LISP family of languages – each list element contains a head and a tail, and a special NIL value marks the end of the list. C programmers can picture this as a linked list. The implementation below derives from Alexandrescu's Loki library, but has been adapted to make use of new C++11 constructs such as variadic templates and template alias types. (Alexandrescu, 2001)

The `utils::typelist` convenience class used in this paper is defined using the following set of templates:

```
template <class...>
struct typelist;

template <>
struct typelist<> {};

template <class T>
struct typelist<T> {
    using Head = T;
};

template <class T, class... Rest>
struct typelist<T, Rest...> {
    using Head = T;
    using Tail = typelist<Rest...>;
};
```

Additionally the following list manipulation functions were implemented:

Table 3. Compile-time list manipulation convenience functions.

Function Name	Explanation
`typelist_cat`	Concatenate two given lists
`typelist_select`	Select a list element at the given index
`typelist_left`	Select the first n elements of the list

These functions were implemented as follows:

```
template <class, class>
struct typelist_cat;

template <std::size_t, class>
struct typelist_select;

template <std::size_t, class>
struct typelist_left;

template <class... Left, class... Right>
struct typelist_cat<typelist<Left...>, typelist<Right...>>
{
    using type = typelist<Left..., Right...>;
};
```

```
template <std::size_t N, class Head, class... Tail>
struct typelist_select<N, typelist<Head, Tail...>>
{
    using type = typelist_select<N-1, typelist<Tail...>>;
};

template <class Head, class... Tail>
struct typelist_select<0, typelist<Head, Tail...>>
{
    using type = Head;
};

template <std::size_t N, class Head, class... Tail>
struct typelist_left<N, typelist<Head, Tail...>>
{
    using type = typename typelist_cat
        < typelist<Head>
        , typename typelist_left< N-1
                                , typelist<Tail...>
                                >::type
        >::type;
};

template <class Head, class... Tail>
struct typelist_left<0, typelist<Head, Tail...>>
{
    using type = typelist<>;
};

template <>
struct typelist_left<0, typelist<>>
{
    using type = typelist<>;
};
```

The usage of these functions was further simplified using a set of alias templates:

```
template <class TypeListLeft, class TypeListRight>
using typelist_cat_t =
typename typelist_cat<TypeListLeft, TypeListRight>::type;

template <std::size_t N, class TypeList>
using typelist_select_t = typename typelist_select<N, TypeList>::type;

template <std::size_t N, class TypeList>
using typelist_left_t = typename typelist_left<N, TypeList>::type;

template <std::size_t N, class... Args>
using select_nth_t = typelist_select_t<N, typelist<Args...>>;

template <std::size_t N, class... Args>
using left_n_t = typelist_left_t<N, typelist<Args...>>;
```

Together these templates provide sufficient expressiveness to work with arbitrary lists of types at compile time. In particular, it is possible to store types using the definitions above for later processing, which is not possible with native C++ parameter packs.

Appendix D. utils::invoke_tuple utility

The current C++ standard does not define a utility function for calling arbitrary functions using an unpacked tuple. However, such a function is relatively straight forward to write:

```
template <class Func, class... Args, std::size_t... Is>
inline auto
invoke_tuple(Func &&f,
             std::tuple<Args...> &&args,
             std::index_sequence<Is...>)
-> decltype(
     std::forward<Func>(f)(
       std::forward<Args>(std::get<Is>(args))...))
{
  return std::forward<Func>(f)(
    std::forward<Args>(std::get<Is>(args))...);
}

template <class Func, class... Args>
inline auto
invoke_tuple(Func &&f, std::tuple<Args...> &&args)
-> decltype(
     invoke_tuple(std::forward<Func>(f),
                  std::forward<std::tuple<Args...>>(args),
                  std::index_sequence_for<Args...>{}))
{
  return invoke_tuple(std::forward<Func>(f),
                      std::forward<std::tuple<Args...>>(args),
                      std::index_sequence_for<Args...>{});
}
```

A similar function named `std::apply` has been proposed for inclusion in the C++ standard library in the future, but has not yet been ratified. (ISO/IEC, 2014b)