Transmorphic: Mapping direct Manipulation to Source Code Transformations

Robin Schreiber, Robert Krahn, Daniel H. H. Ingalls, Robert Hirschfeld

Technische Berichte Nr. 110

des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam | 110

Robin Schreiber | Robert Krahn | Daniel H. H. Ingalls | Robert Hirschfeld

Transmorphic

Mapping direct Manipulation to Source Code Transformations

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de/ abrufbar.

Universitätsverlag Potsdam 2017
http://verlag.ub.uni-potsdam.de/

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 2533 / Fax: 2292
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam
URN urn:nbn:de:kobv:517-opus4-98300
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus4-98300

Zugleich gedruckt erschienen im Universitätsverlag Potsdam:
ISBN 978-3-86956-387-9

Defining Graphical User Interfaces (GUIs) through functional abstractions can reduce the complexity that arises from mutable abstractions. Recent examples, such as Facebook's React GUI framework have shown, how modelling the view as a functional projection from the application state to a visual representation can reduce the number of interacting objects and thus help to improve the reliabiliy of the system. This however comes at the price of a more rigid, functional framework where programmers are forced to express visual entities with functional abstractions, detached from the way one intuitively thinks about the physical world.

In contrast to that, the GUI Framework Morphic allows interactions in the graphical domain, such as grabbing, dragging or resizing of elements to evolve an application at runtime, providing liveness and directness in the development workflow. Modelling each visual entity through mutable abstractions however makes it difficult to ensure correctness when GUIs start to grow more complex. Furthermore, by evolving morphs at runtime through direct manipulation we diverge more and more from the symbolic description that corresponds to the morph. Given that both of these approaches have their merits and problems, is there a way to combine them in a meaningful way that preserves their respective benefits?

As a solution for this problem, we propose to lift Morphic's concept of direct manipulation from the mutation of state to the transformation of source code. In particular, we will explore the design, implementation and integration of a bidirectional mapping between the graphical representation and a functional and declarative symbolic description of a graphical user interface within a self hosted development environment. We will present Transmorphic, a functional take on the Morphic GUI Framework, where the visual and structural properties of morphs are defined in a purely functional, declarative fashion. In Transmorphic, the developer is able to assemble different morphs at runtime through direct manipulation which is automatically translated into changes in the code of the application. In this way, the comprehensiveness and predictability of direct manipulation can be used in the context of a purely functional GUI, while the effects of the manipulation are reflected in a medium that is always in reach for the programmer and can even be used to incorporate the source transformations into the source files of the application.

Contents

1. Introduction

In this work we will explore the design, implementation and use cases of Transmorphic, a functional interpretation of the Morphic GUI Framework, where the visual and structural properties of morphs are defined in a purely functional and therefore declarative fashion. The name *Transmorphic* is derived from the idea of *transforming a morph* to different representations, each of which provides its own set of benefits for the programmer.

1.1. The Morphic GUI Framework

Morphic [24] is a framework for describing graphical user interfaces (GUI), which was initially introduced in the programming language Self [5]. What distinguishes Morphic from other GUI frameworks is that it provides a high degree of directness and liveness when developing an application with a GUI. *Directness* refers to the possibility for a designer to directly refer to the graphical representation of an interface and thereby change its visual, structural or behavioral properties. On the other hand the *liveness* of Morphic guarantees that any adaptations at runtime are immediately reflected in the running program. This includes changes in the symbolic description of a program or manipulations of the GUI elements.

To provide these properties, Morphic represents each visual entity in the GUI by an object that combines the object specific behavior, state and visual specifications in itself. These objects are referred to as *morphs* and can be referenced and altered in the symbolic domain through textual editing, or alternatively in the visual domain through direct manipulation by means of a tool called the *halo* [31]. Morphic is able to combine these two different views (visual and symbolic) onto a graphical user interface, allowing people from different backgrounds (i.e. designers or computer programmers) with different approaches (i.e. visual or abstract reasoning) to work on the same program. Also, Morphic does not distinguish between development modes such as run-time or edit-time and remains fully adaptable throughout execution. For this reason, Morphic is mostly used in the context of live development environments, such as Squeak/Smalltalk [19], Self [5] or Lively Kernel [33].

Morphic's approach is appealing due to its close resemblance to the physical world: Being able to directly manipulate individual objects at runtime aligns closely with many people's intuitive understanding of a rendered GUI.

1.2. Functional Graphic User Interfaces

Mutable abstractions are a very powerful concept in computer programming, since they are able to closely reify concepts from the real world and are often very efficient with regards to memory and computation time. However, this power also comes with a great responsibility, and if used carelessly can significantly increase the complexity of an application.

Contrary to that, in a purely functional program, state is managed entirely via the parameter values of the function calls, and never by mutation of information that is stored behind an address, outside of the execution stack of the program. This property is also called *referential integrity* or *value based programming* and ensures that the values that a functional program works with are *immutable* at any point in execution.

It is important to understand that immutability by itself does not protect from the complexities of a mutable abstraction, since errors caused by operations that do not commute or faulty updates are still possible. Based on this *explicit* way of managing state, we are, however, able to derive new abstractions that can *replace* the existing mutable abstractions in order to make state more manageable.

To clarify this, let us take the problem of synchronizing controller and view in the MVC design pattern [22]. There are different approaches to implement this scenario based on mutable abstractions (i.e. via an observer or mediator pattern). However in any of these cases we are faced with the constant challenge to *replicate* any change that happens inside the model or controller also within the view in order to keep both representations of information synchronized. The more complex model or view become, or the more controller and view objects start to interact with each other, the more effort we have to put into synchronizing both entities. However, if we instead consider the view to be a *functional projection* [11] from the model to the rendered view, the situation starts to look different: Now that every element is explicitly derived from a value inside the model, we no longer need to implement a separate update mechanism. Instead, we now just evaluate the functional projection together with the updated model and retrieve an updated view without having to provide a separate update mechanism. In addition to that, the code for the GUI becomes more declarative because the relationship between values and visual entities is purely functional.

In the context of GUI programming, the functional approach can provide us with a more stable view implementation that also expresses the intent of the view more clearly, thereby making it more understandable.

1.3. Lifting the Concept of Direct Manipulation

Both Morphic and the functional approach to rendering a GUI have their respective benefits and liabilities. We argue, that a combination of a functional view description and the direct manipulation interface of Morphic creates benefits for experts and beginners alike: From a professional GUI developer's perspective a combination of

both sides would provide the fast prototyping capabilities of Morphic, together with a functional foundation of the rendered GUI, that helps to reduce the complexity inside the application. On the other side, novice programmers, who are new to functional programming, can use Morphic's direct manipulation interface to get a helping hand when taking the first steps in a system, where views are described in a purely functional manner. This combination of programmatic and symbolic manipulation is also referred to as *Prodirect Manipulation* [6].

Existing implementations of Morphic build on the fact that the *gestalt* of a certain morph is represented by mutable state that we can manipulate through direct manipulation. The problem which remains is *how* a direct manipulation interface should be implemented in the context of a system where the representation of the GUI is essentially free of state.

As a solution to this problem, we propose to lift the concept of direct manipulation from the mutation of state to the transformation of code. By that, we mean that the code that is responsible for the rendered graphical representation is changed and newly evaluated, such that the direct manipulations are reflected accordingly in the graphical representation.

1.4. Contributions

Our proposed solution is based on combining the benefits of a declarative, tree-based description of a morphic scene graph and the object-oriented interpretation of a morph, where each visual entity comes with an identity and can be manipulated directly. We will present Transmorphic, which is a functional version of the Morphic GUI Framework that emphasizes a clean separation between code that is responsible for stating visual and structural properties and code that manages the model of the application. In Transmorphic, morphs are defined in an entirely functional and therefore declarative fashion, where visual properties of a morph scene graph are functionally derived from the application state. The interface remains responsive, by automatically triggering a complete reëvaluation of the functional view once the application state has been modified. Application state is evolved by action handlers that are collocated with the declarative description of visual properties inside the symbolic description. Direct manipulation changes are mapped to transformations in the symbolic description of the application, such that afterwards the rendered morphs match the enacted manipulation. To provide a symmetric relationship between the symbolic description and the visual representation, Transmorphic makes use of functional lenses [2] which allow an operation to be translated to different yet homomorphic data structures. We are therefore able to constantly reconcile graphical representation and symbolic description of a morph whenever the user applies a transformation of the morphic scene graph by direct manipulation.

In particular, the contribution are as follows.

1. Presentation of the concept behind Transmorphic, where morphs are defined in a purely functional way.
2. Explanation of Transmorphic's implementation.
3. Comparison of Transmorphic to other implementations of Morphic by means of a concrete use case scenario.

Section 2 will provide the technological foundations of Transmorphic, while Section 3 will present Transmorphic and the concept behind the provided functionalities. Secion 4 will describe the implementation of Transmorphic and the challenges we faced when building a GUI system on functional abstractions. We will then present an example workflow in Transmorphic and compare it to two alternative workflows in Lively Kernel and Squeak/Smalltalk in Section 5 while also discussing the limitations and future work with regards to Transmorphic. Section 6 will present related work. We will finally summarize our findings in Section 7.

2. Technological Foundations of Transmorphic

Transmorphic is directly derived from the Morphic GUI framework which itself has been implemented multiple times based on different interpretations. As a system, Transmorphic combines ideas from the functional world (*Functional Lenses*, *Immutable Data Structures*) and systems that provide direct manipulation authoring (*Halo*, *Object Editor*) in a way that enables the programmer to evolve the code and the corresponding graphical representation simultaneously.

Transmorphic is built in the programming language Clojure [16] which provides us with powerful mechanisms to instrument and transform code at runtime, and serves as a base for Transmorphic's source code transformations. We further used a version of Clojure, called Clojurescript [26], that compiles to Javascript and allows us to run in the context of the web browser. This makes Transmorphic accessible from a wide range of platforms, not requiring any kind of installation routines that need to be performed by the user beforehand.[1]

2.1. Immediate Mode and Retained Mode Rendering

In the domain of computer graphics, we differentiate between two different modes in which a scene graph can be rendered: One approach is to define the scene graph by means of primitive elements that are composed together and are then managed on behalf of the runtime. We refer to this as *retained mode rendering* [28], since the scene graph is reified by a composite of objects which are retained by the runtime. Morphic simplifies the implementation of a direct manipulation interface mostly from the fact that it is based on retained mode rendering, where adaptions to the scene graph can be performed *incrementally* at any point in time.

The other approach requires us to keep the structure of the scene graph in a separate, possibly more abstract representation and perform rendering by explicitly calling functions that correspond to visual elements to be displayed. In this mode, each time a frame is drawn the application needs to issue a function call for every visual element to be displayed, essentially rebuilding the visual representation from scratch. A concrete example for this so called *immediate mode rendering* is the Doc-

[1] Due to current technical limitations, we require a locally running *Java Virtual Machine* (JVM) next to the browser. However conceptually, this is not necessary, and future implementations of Transmorphic could very well be executed completely independent of the JVM.

ument Object Model (DOM) which is the internal representation of the rendered document inside a web browser. Here, each node inside the DOM is interpreted as a function call to display a certain element inside the browser. When we change a node inside the DOM, the browser renders a new frame in order to update the graphical representation of the document. However, modern client-side programming, namely *dynamic html* [14], has started to treat the DOM more as a piece of mutable data, rather than an immutable *immediate mode* description of the document that has been processed by the web browser. This has resulted in the peculiar situation of an *immediate mode* description (the DOM) being used in a way that is almost identical to that of *retained mode rendering* (a mutable abstraction).

Another example where retained and immediate modes are combined with each other is the use of *shaders* [10] in order to perform adaptations to the (functionally) rendered vertices or pixels in an incremental manner. We will see that Transmorphic also combines both modes in order to reconcile a functional view description with a direct manipulation interface.

2.2. React and the Merits of Functional Programming

The GUI Framework *React* [11] is the result of a major engineering effort by Facebook to decouple the browser's DOM from the imperative programming paradigm. Over time, the rendering engine in Facebook's various web applications had grown so complex, that the UI repeatedly started to diverge from the actual data that was supposed to be rendered. The large number of collaborating view and controller objects forced the engineers at Facebook to reconsider their approach of how data was rendered in their applications. The usual way to implement interactive graphical interfaces in the context of the client is to associate groups of nodes inside the DOM with application data, and then updated the nodes as necessary to reflect the changes. With React, the intention was to change this way of rendering data to the DOM by replacing the explicit management of DOM nodes with a functional projection instead. This essentially means, that the *retained mode* of managing the DOM gets replaced by an *immediate mode*, where visual entities correspond to functions, and we get to render a scene by repeatedly calling these functions. We end up with a declarative way to define the view of an application, where the order or permutations of operations that we initially required to synchronize view and controller, are no longer part of the picture. Instead, the "mutable part" of application is extracted to action handlers that are triggered in response to user actions and then transform the application state accordingly. Once changes in the application state are performed, the immediate mode render pass is triggered again, causing the view to update accordingly.

Naively implemented, this approach would work, yet result in unresponsive user interfaces, since the numerous alterations to the DOM involved with every re-rendering of the view and would reduce the performance: The consolidation of the DOM, that happens after a mutation, is referred to as "reflow". The reflow is a user-blocking operation, may require a lot of time to compute, and can therefore lead to

confusing degradation of performance when using the DOM naively. For example, the mutation of DOM nodes that are deeply nested inside the document tree, or their attributes, can become surprisingly slow and lead to a significantly reduced responsiveness of the running application. In order to keep a web application responsive, programmers have to carefully mutate the DOM in order to minimize reflow times. This restriction has made it especially difficult to introduce abstractions that ignore this nature of the DOM. For example, a purely functional description of the graphical user interface being rendered in the client may very often lead to large and unnecessary amounts of updates in the DOM simply because the notion of different mutable parts that the DOM consists of, is not part of a functional picture of the world [18]. React therefore comes with an algorithm that determines the differences between DOM and the HTML to be rendered, and translates the difference to the smallest set of operations necessary to adapt the DOM to fit the new version of the view. Since the diff processing happens in pure Javascript, the impact on the performance is fairly small, although the problem at hand is a fairly complicated one which needs advanced heuristics to perform in reasonable time.

Despite its name, React is actually not an implementation of Functional Reactive Programming (FRP) [40] or Functional Reactive Animation (FRA) [9]. In FRP and FRA, user inputs and other sources of information are modeled as signals that change over time. Changes in these signals can be triggered by actions from the user or other sources outside of the application. A change will then trigger a chain of function calls that eventually result in a new end value, which is then used to render the new state of the application. This is not the case in React, it instead borrows the reactive part from FRP, meaning the phase when the functional program reacts to the changes that happened in the input signals of the application. This reactive part is then embedded in what is just a classical, imperative application, using callbacks to trigger changes in the data and evolve the application. Although React only replaces a part of the system that was previously managed through object collaboration, this measure causes many of the existing UI consistency bugs to disappear. This is because the functional description of views in React manages to defines both the graphical representation, as well as the reactive update mechanism. By replacing something that was previously a view object with a functional derivation of the application's state to visual elements, we remove the need for mutable abstractions in the view. From an engineering point of view, we think that it is this very removal of mutable abstractions that makes React less prone to errors when providing a consistent view in user interfaces.

The programming model of React will not be described in detail here, since this would exceed the scope of this work. We should however note, that we borrowed various concepts from the React programming model, such as *Components* or *unidirectional data flow*, when designing Transmorphic. Simply put, Transmorphic is similar to React with the difference that (1) we render morphs instead of HTML tags, and (2) we have added support for direct manipulation to evolve the application.

2.3. Functional Lenses

In order to model changes to immutable data structures, we need to represent these as first class entities that operate on the data structure as a whole. For example, in the case of a deeply nested dictionary, we need to supply a list of keys, *always starting from the root*, that leads to the datapoint we want to change, together with the new value that should take its place. This *explicit* notion of a change inside a data structure can also be generalized, and we can develop strategies that allow us to apply the *same* change to *different* data structures in case there exists a homomorphism between the data structures. For example, a hash map in Clojure can either be represented by an XML document, a JSON document or in the standard notation for data structures used inside Clojure, namely *Extensible Data Notation* (EDN). Given that we can generalize changes over the different layouts of the data structures, it should be possible to directly project changes that happen to one of these representations to all of the other ones: Supposing we change the JSON representation and determine a key is changed, we can implement a mechanism that finds the corresponding part within the XML and changes the value there accordingly.

In functional programming, we call abstractions that provide this kind of change mapping between data structures, *functional lenses* [27]. A lens is always defined between a pair of data structures (i.e. JSON/EDN, JSON/XML, etc…) one of which we call the *target* and the other, which we call the *source*. The roles of these data structures is important: Since lenses usually do not provide a bijective mapping, the source data structure will contain the complete set of information, while the target data structure is often a more simplified or reduced representation of the information. Each lens itself consists of a pair of functions, namely `GET` which allows us to retrieve the representation of the source data in the target data structure and `PUTBACK` that is used to update the source data by providing an updated version of the target data.

```
GET(source) -> target

PUTBACK(target*, source) -> source*
```

For a lens that maps changes from JSON to XML, the formula could look more like this:

```
GET(source.json) -> target.xml

PUTBACK(target*.xml, source.json) -> source*.json
```

Also notice that we are able to compose lenses with each other, which often frees us from the need to redefine a lens for every pair of data structures we encounter. When working with lenses in general, and especially in the case of composition, it is important to note, that *not all lenses are created equal*. In fact, we are able to classify different lenses with regards to "how well" they are able to establish a bidirectional

relationship as seen in Figure 2.1.[2] This means, that in the case of composition, the resulting macro lens can only be as good as its "worst" sub lens.

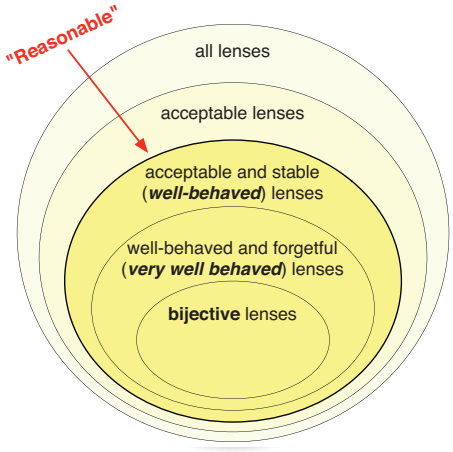

Figure 2.1.: The lens classification hierarchy

The "lowest" class of lenses we can deem "acceptable" are the ones that correctly translate updates back and forth from the target domain to the source domain. More formally, this means that any `PUTBACK` that is immediately followed by a `GET` will *always* return the same target value:

```
GET(PUTBACK(target, source)) = target
```

An example for an "unreasonable" lens would be one where the `PUTBACK` function were to always return the same value. In this case, we would usually never retrieve the same target we put back into the source before.

The next best class of lenses we call "well behaved", which is when they are stable, meaning that when the target is not modified, we also do not modify the source in case a `PUTBACK` happens:

```
PUTBACK(GET(source), source) = source
```

For example, this excludes all lenses which perform some kind of bookkeeping inside the source each time a `GET` is performed, such as incrementing a counter. Put differently a lens is considered well behaved if and only if its GET function is pure. While the well behaved lenses constitute the majority of lenses we use in practice there are further classes of even "stricter" lenses which become more and more difficult to satisfy.

[2]Copyright 2006 Benjamin C. Pierce, Microsoft Research, Cambridge https://www.cis.upenn.edu/~bcpierce/papers/lenses-etapsslides.pdf (slide 71), last accessed 2016-04-14.

This brings us to the next higher class of lenses, the "very well behaved" ones, where in addition the `PUTBACK` is "forgetful" and will always *completely* overwrite the previous update:

```
PUTBACK(target_2, PUTBACK(target_1, source)) = source_2

GET(source_2) = target_2
```

Notice that while this property may seem sensible, it also restricts us in some ways. For example, lenses that are not forgetful make it easy to quickly undo or redo changes, since the information about previous `PUTBACK` calls may always be preserved.

The last class, and most restrictive for that matter, are the lenses that form a *bijective* mapping between source and target data structure. These are fairly rare, especially in the context of graphical representations, since a view is usually discarding information during the process of rendering information. In order to be bijective, we must not discard information whenever we `GET` or `PUTBACK` information, and both representations need to be able to store the same amount of information in order to transmit it back and forth.

2.4. Clojure

In order to ensure control over how state evolves in Transmorphic and to be able to instrument code at runtime in arbitrary ways, we decided to implement Transmorphic entirely in Clojure [16], which is a dialect of LISP that runs on top of the JVM.[3] Clojure is one of the most recent LISP dialects, that was first released in the year 2005 by Rich Hickey. The creation of Clojure arose from Hickeys dissatisfaction with the Java programming environment that he was working with at his daily job. These circumstances eventually lead him to create a flavor of LISP, that would compile to JVM bytecode, run on top of the JVM and was able to interface with arbitrary Java libraries and free the developer from having to deal with Java and its abstractions directly. Clojure is opinionated about the way the programmer should organize the data of an application, in that it tries to leave the stored data of a program as unadorned as possible and also keeping the amount of mutable state to a minimum. Similar to the way Self justified its prototype-based approach, Clojure argues that any kind of model or object oriented abstraction eventually turns out to be "wrong" or "insufficient" and from then on, starts to "obscure" information and hinder the development of the application rather than supporting the developer [17]. For this reason, all language constructs in Clojure are non invasive, in that they never en-

[3] Since this would exceed the scope of this work, we will not give a general introduction to Clojure and assume that the reader is already familiar with this language. People that are new to Clojure, we refer to the well written Clojure documentation, available online at: https://clojure.org/.

force a structure upon the data they operate on, but rather interpret data in custom defined ways and fail on their behalf if the data does not provide the necessary information. For example, to implement polymorphism, Clojure favors the approach of multimethods instead of subclassing and overriding the respective methods of an object [15]. In the following sections, we will describe some of the language features of Clojure, that are especially important to understand the inner workings of Transmorphic.

2.4.1. Persistent Data Structures

To minimize the amount of mutable state in the application, Clojure emphasizes the widespread use of immutable values and urges to perform a majority of computation without mutating variables of any kind. In the context of data structures that embody collections, refraining from mutation essentially means that a new version of that data structure has to be returned any time an adjustment is applied to the stored information. Data structures that adhere to this specific property are called "Persistent Data Structures" since previous references still reference the state of the data structure when it was initially assigned.

Since mutable state should be reduced to a minimum, persistent data structures play a central role in most computations in Clojure, making it vital that they are implemented in a memory and performance efficient way. For this reason, all of Clojure's persistent data structures (namely lists, sets, vectors and hashmaps) are implemented in almost the same way by a Hash Array Mapped Trie [1]. Technically, a trie is able to store multiple versions of a certain data structure in itself, while keeping the amount of redundant information to a minimum. In a trie, the hash that corresponds with a stored value is always also the *location* that datapoint is stored at. When a new version of the trie is supposed to be created, the trie takes the hash that retrieves the current version of the data structure, identifies the location where the new version differs from the previous one, and inserts a new entry into the trie to store the new bit of information. An updated version of the hash is now returned, which is used by the new reference to now point to that "new" version of the persistent data structure. Meanwhile, the old value can still be retrieved by the previous hash, making the implemented data structure seemingly immutable by only using a small set of mutations behind the scenes.

2.4.2. Atoms

There is always a point in time where a computation needs to communicate its result with other computations. While immutable values simplify the reasoning about state, and therefore spare our complexity budget when building an application, we need the ability to mutate state in a Turing Machine based model of computation in order to communicate our information to other processes. In the context of Turing Machines, Mutable state is therefore a necessity in order for programs to be of any use, but it demands alertness from the programmer, since the introduction of side effects can quickly grow out of hand. In order to control the scope of side effects, we

need to control when and where they happen, for example by restricting the access to mutable state. How this is achieved in the best way, is answered differently by each programming language: For example, the language C [21] enforces little control over mutations, allowing any datapoint accessible in the current scope to be altered at any time. By using the `const` keyword, the programmer is able to declare functions free of side effects, which the compiler can use to ensure that other functions declared `const` actually satisfy this property. Note however, that the compiler can not guarantee that a `const` declared function is actually free of side effects so the programmer is still responsible for correctness of the program after all. Higher-level languages, such as Java, Python [38] or Smalltalk [13] try to contain the complexity of mutable state, by restricting access to mutable values in various ways: Java for example, allows to control the access to mutable state by declaring variables public or private, Python prevents variables declared global to be mutable at all, and Smalltalk by default does not provide direct access to instance variables held by an object. There are of course more drastic measures to contain side effects, as encountered in the functional programming language Haskell [35], which does not even provide a language primitive to introduce side effects at all. Haskell either replaces the need for a side effect, by threading the state through a functional abstraction called a "Monad" or resorts back to externally implemented libraries, that then perform the required side effect on their behalf [39]. Due to this rigorous measure, one very frequently used external library in Haskell is the I/O-Monad, that provides a monadic interface for communicating with external processes, encapsulating all mutation of state within its implementation.

Clojure's approach of managing mutable state does not abstain from mutating values entirely but rather allows the programmer to precisely denote places where it is possible for state to be mutated. Clojure achieves this by wrapping any immutable persistent data structure or primitive value inside a special container called an "atom". An atom is itself a value that can be dereferenced to return the currently stored value inside the atom. By restricting the access to the actual value inside an atom through a function call, it is possible to swap or replace the current value stored inside the atom in a transactional context. This design has various implications: For one, we are able to guarantee a consistent view onto the value of an atom since all operations that mutate the atom's value are atomic making it impossible to ever read an inconsistent bit of information. Secondly, the number of side effects is usually reduced by defining computations in a functional way, relying on persistent data structures and then perform a single transaction to persist the result. Lastly, atoms enable us to instrument the transactional context in arbitrary ways, allowing us to track how the state inside the application progresses over time. This makes it easy, to revert state to earlier stages or to implement mechanisms that observe a certain datapoint inside an application, which simplified the implementation of Transmorphic.

2.4.3. Code is Data

The phrase "Code is Data" comes from the family of LISP programing languages and refers to LISP's rather unique way of treating source code in that same manner as plain data. In any LISP, the translation of code to an executable format, is actually split in two separate phases: The reader phase and the actual compilation phase. In the reader phase, code is parsed from a source file into the internal list structure of the LISP runtime, effectively turning the textual representation of the code into LISP's internal schema to store all data [25]. Given that we have all the code in the form of data in memory, we are able to define special functions that are invoked at compile time, which are able to operate on the list structures in arbitrary ways. LISP refers to these special functions as macros, and they can be used to transform and instrument code (actually data) in arbitrary ways making LISP a formidable environment for implementing domain specific language constructs. Whenever the compiler encounters a symbol at the start of a list, that refers to a macro, it invokes that macro passing the rest of the list as an argument, and then replaces the cons cell that stores the symbol with the return value of the macro. During macro expansion, the macro functions have access to a fully-fledged runtime, having no restrictions in the types of operations or libraries that can be used. Obviously, transforming code in this manner is not restricted to compilation, but can be used to assemble and evaluate symbolic expressions at any point in time during the execution of the actual program. Being a fully compliant LISP dialect, Clojure provides this macro system, which is used by Transmorphic in order to transform symbolic descriptions at runtime and compile time.

2.4.4. Clojurescript

While Clojure provides powerful abstractions to reason about state and transform code, it fully relies on a locally running JVM and can therefore not be directly executed in the context of a web browser. Fortunately, Clojure is designed to be compiled to various backends and since 2010 exists in a version that can be compiled to Javascript, called Clojurescript [26]. Transmorphic is therefore written entirely in Clojurescript, which allows us to leverage all of Clojure's state management and code transformation facilities. From a technical perspective, the notion of code as data makes it very easy to compile Clojure to different platforms, since we essentially just have to replace the step that initially yielded Java Bytecode to now produce Javascript statements instead. One thing to keep in mind though is that Clojure's seamless interfacing with other Java libraries makes existing Clojure code not as portable as it might seem at first. Thus many modules need to be rewritten in case they have dependencies to parts of the Java standard library. As of today, the majority of Clojure is fully compilable to Clojurescript but despite being able to run in the context of the browser, the compilation steps mostly still require a JVM to be present during compilation. In particular, many macros in Clojurescript still employ functions of the Java standard library, since they are only invoked at compile time not taking into account that one may also want to compile code within the Clojurescript

(i.e. Javascript) runtime. Clojurescript currently provides the compilation of Clojure code in the context of the browser at an experimental stage, but is still too incomplete to be used as the basis of a self hosted development environment. For the time being, Transmorphic therefore requires an externally running REPL that is deployed on top of a JVM where the majority of runtime compilation is dispatched to. Development of Clojurescript has been very active though, and the number of developers using Clojurescript has already surpassed the number of active Clojure developers for the JVM, which gives rise to hope that in the near future Clojurescript will be fully self hosted, obviating the need for a separately running JVM.

3. The Transmorphic GUI Framework

Transmorphic combines a functional and declarative way of describing GUIs with the notion of a morphic scene graph, where each rendered morph can be uniquely identified and also modified. In this chapter, we will first describe the symbolic building blocks, and conceptual model of Transmorphic and then take a look at how we are able to introduce direct manipulation into this system. Transmorphic comes with two different fundamental abstractions, called *Morphs* and *Components*, both of which we will refer to as *entities* when we do not have to distinguish between the two. We will further take a look at the different tools that allow the programmer to integrate direct manipulation effectively in the context of Transmorphic.

3.1. Morphs

Morphs are the core building blocks that make up the description of a GUI in Transmorphic. They are atomic, indivisible entities that are stateless and represent different kinds of visual entities such as rectangles, images, ellipses or polygons. In Transmorphic, a morph is created by a function call that is being passed a key-value map of properties followed by an arbitrary number of additional morphs that will constitute the set of submorphs. For instance, if we want to render an image morph, we call the function `image` as in Listing 3.1.

Listing 3.1: Example for a function call in Transmorphic that yields an image morph

```
1 (image {:extent ($parent :extent)
2         :position {:x 42 :y 42}
3         :url "kermit.png"
4         :on-mouse-enter (fn [e]
5                           (prn "Mouse entered!"))})
```

Looking at this example, we can separate the kinds of properties that can be passed to a certain morph into three different types:

Primitive Properties These are properties that define the visual appearance of a morph, such as its fill, extent or position. Also there may be certain properties that only apply to some morphs, for example in this case the `:url` property will define the path to the image being displayed inside the image morph. The

values of primitive props are either strings, numbers or collections that satisfy a certain schema. For example the `:extent` property expects its value to be a collection in the form of a hash map that contains `:x` and `:y` keys that each reference a number.

Behavioral Properties Include callbacks, to certain mouse or keyboard events. Also includes specific morphic events such as grabbing or dragging of a morph. Behavioral properties present the entry point to perform changes inside the application state, which eventually causes the graphical representation to update.

Relative Properties Primitive properties may actually be defined in terms of other morph's properties in the current morphs surroundings. Defining properties relative to one another, simplifies composition of morphs, by obviating the need to pass the values of certain extensional properties explicitly.

The hash map of properties can contain arbitrary key value pairs, yet only the ones recognized by the actual implementation of the morph, will have an effect.

In Morphic, composition is key to creating a final interface that provides directness and liveness for the developers of the application. Taking the previous example again, we can wrap additional morphs inside the image morph we have already introduced, by composing the function calls that yield the respective morphs together, as seen in Listing 3.2.

Listing 3.2: Submorphs are added to another morph through function composition. This is an example of an `image` morph that contains two submorphs, namely an `ellipse` and a `rectangle`.

```
(image {:extent ($parent :extent)
        :position {:x 42 :y 42}
        :url "kermit.png"
        :on-mouse-enter (fn [e]
                          (prn "Mouse entered!"))}
         (ellipse {:position {:x 0 :y 0}
                   :fill "green"
                   :extent {:x 100 :y 100}})
      (rectangle {:position {:x 42 :y 42}
                  :fill "red"
                  :extent {:x 100 :y 100}}))
```

The submorphs of a certain morph are automatically flattened by the runtime of Transmorphic, so there is no need to keep track of nested collections of morphs. For example we can easily add a collection of morphs which is computed through a map statement and not have to worry about flattening the submorph collection passed to the image morph, as seen in Listing 3.3.

Listing 3.3: Submorphs can also be defined in terms of collections, such as this map statement. To save space, we excluded parts of the property definitions.

```
(rectangle { ... }
   (image { ... })
   (rectangle { ... })
   (map (fn [i]
         (text {:position {:x (* 10 i) :y (* 10 i)}
                :value (str i)}))
          (range 10)))
```

We refer to the morph that contains a set of other morphs as the *parent* of these morphs. Notice, that we can also render submorphs conditionally and do not have to worry about nil values causing trouble since the flattening mechanism of Transmorphic automatically weeds out these as well. In the symbolic description of a morph scene graph, immutability is a key property, which means that without explicit use of meta programming facilities, no mutations to neither properties nor structure of a morph can be introduced once it is created.

3.1.1. Defining new Morphs

A morph is always defined in terms of HTML elements, which gives the programmer maximum flexibility in terms of defining the appearance of the respective morph as shown in Listing 3.4.

Listing 3.4: Example of a morph definition. The helper method `html-attributes` ensures that the properties passed to the morph are correctly translated to style properties and javascript callbacks respectively.

```
(defmorph image
  [{:keys [morph-id type props submorphs]}]
  (apply div (html-attributes props morph-id)
    (div nil
      (img (clj->js {:style (shape-style props)
                     :src (props :url)}))))
       (build-all render-morph submorphs)))
```

A morph, once defined and used, may be varied by passing different props, and composed together with other morphs; however, it can not be decomposed any further since it constitutes an atomic building block. Furthermore, the implementation of a new morph, passes several responsibilities to the programmer: For one, it must be ensured that the behavioral properties are correctly installed into the HTML

elements the morph consists of. In addition to that, tools, such as the halo, that provide an interface for direct manipulation, assume that certain properties such as *position*, *extent* or *rotation* are correctly recognized by a morph's implementation. Transmorphic already comes with a variety of predefined morph types, which includes rectangle, ellipse, image, polygon, text and a html morph that can be used to embed arbitrary html fragments into the morph hierarchy and allows to interface with external javascript libraries such as code editors embedded in HTML. Due to these reasons, it is encouraged to keep the number of customly defined morphs to a minimum and stick to composition instead of defining new morphs to implement a certain interface.

3.2. Components

While morphs are able to cover all of the visual aspects of the UI, they are entirely free of state and in many cases too fine grained to serve as a reusable abstraction. Also, we have not yet provided a mechanism that allows us to actually render a morph scene graph or manage state in a more modularized fashion. For this reason, Transmorphic provides the programmer with the ability to group morphs into components. A component can be summarized as an abstraction that reifies a reusable element inside the GUI of an application. To the outside, a component appears just like a morph in that we create it by calling a component function together with a set of properties followed by all the submorphs, as shown in Listing 3.5.

Listing 3.5: Example of a component `hour-pointer` being rendered as a submorph of an ellipse morph

```
(ellipse
  { ... }
  (hour-pointer
    {:radius radius, :hours (-> time :hours)}
    (rectangle ...)))
```

Things start to get more interesting once we look at the definition of a component as seen in Listing 3.6.

The return value of the render method always needs to be either a morph or a component which we also refer to as the "root" of a component. The structure of this root entity can be influenced by means of three arguments that are always passed to the render method of a component once it is getting rendered:

The first argument (usually denoted by *self*), is a reference to the component local state that we use as the first argument to all the functions that allow us to read and

Listing 3.6: Example of a component definition

```
(defcomponent hour-pointer
  IRender
  (render [self props submorphs]
        (rectangle
         {:id "HourPointer"
          :position {:x 0 :y -2.5}
          :rotation (* (+ -0.25 (/ (props :hours) 12)) PI 2)
          :fill "darkblue"
          :stroke-width 2
          :extent {:x (* .5 (props :radius)) :y 5}}
          submorphs)))
```

write to a component local state. Reading from and writing to this local state reference works just the same as with normal atoms in Clojure (`swap!`, `reset!`, `deref`) 3.7.

Listing 3.7: The interface to access and transform the component local state is the same as Clojure's atom interface.

```
; dereferencing the component state, returning its value:
@self
; update the value inside the component state by means of a function:
    ↪
(swap! self assoc :key :value)
; rest the the value inside the component state by a constant:
(reset! self 42)
```

Notice that writing to component local state is turned off throughout the render pass of a component, which means that mutations can only happen within the action handlers that were provided via the behavioral properties.

The second argument (in this case denoted by *props*) is the hash map of properties that was passed to the component. In many cases, this is almost the same set of properties that the root of the component will receive. To relieve the programmer from constantly having to manually merge the properties of the root and the component, Transmorphic by default merges the components properties with the properties of the component's root. The root may still define custom properties that are independent of the ones passed to the component by simply stating them inside the lexical scope of the render method.

The last argument is the collection of submorphs (morphs and components, denoted by *submorphs*) that belong to the component that is being rendered.

In the end, the rendering process can only be influenced by either the global application state, or the respective local state of the component, which results in a unidirectional flow of information from the few places of mutable data to the eventual rendering of the visual representation. To implement a reactive interface, we place code that evolves the component's local state or the global application state inside functions that are part of the behavioral properties of a morph that is part of a component:

Listing 3.8: Example of behavioral properties that manage the application state in response to user events

```
(def app-state (atom {:counter 42}))

(defcomponent hour-pointer
 IRender
 (render [self props _]
       (rectangle
        {:id "HourPointer"
         ...
         :on-mouse-down (fn [e]
                          (swap! app-state :counter inc)
                          (swap! self assoc :color "green")))))
```

Inside the callback, we can mutate the local state which will cause the view to be rendered again and the visual representation to be updated. In case we modify variables that are outside of the local state of a certain component, we need to trigger a render cycle manually in order for the UI to stay consistent. The component local state, together with the globally accessible values (often stored in atoms), constitute what we refer to as the *application state*. While Transmorphic is currently very liberal about how state is organized, components are a useful abstraction that enables the decomposition of application data with respect to the structure that is embodied in the respective user interface of an application.

Finally, components also function as the initial entry point to start rendering the global morph scene graph, which we also refer to as the *world*. This is done by passing a component's name and initial properties and an atom that will be used to manage the internal state of world to the set-root! function. In addition, we also need to specify a certain DOM node, from where the world is supposed to be rendered.

Transmorphic allows the programmer to evolve an arbitrary number of different worlds in parallel, provided that we pass a different world atom for each of them in order to guarantee isolation. Inside the world atoms, Transmorphic stores an internal copy of the symbolic description and the currently rendered scene graph, which allows the different worlds to evolve completely independent from each other.

Listing 3.9: Via `set-root!` the root DOM node from where the morphic world is evolving can be specified.

```
(set-root!
   world
   my-component
   {:id "foo"}
   (gdom/getElement "app"))
```

3.2.1. Owners and Parents

Components constitute a somewhat different structure in our previously purely morph based scene graph since they are not atomic but compositions of other morphs and components. In order to make components fit into the whole picture, we need to introduce the owner relationship in addition to the already described parent-submorph relationship. The owner-ownee relation denotes the connection between a component and the composition of morphs that it consists of. Since the parent of a morph is already its direct ancestor morph, we refer to the component that initially passed the properties to a morph as the morphs owner. A different way of putting this, is to say that a component is the owner of all morphs that appear in the lexical scope of the component's render method. Note that a component can also be the owner of another component, in case it is rendered within the scope of another components render method. Likewise, the parent of a component simply remains the direct ancestor morph that it was placed in to be rendered as part of a submorph. To further clarify the difference between these two orthogonal relationships between components and morphs, take a look at Figure 3.1

3.2.2. Component Lifecycle Protocol

On the one hand, the interface a component provides to the outside is that of a stateless entity, which can be rendered next to other morphs which again are also just pure functions. Since view objects in Transmorphic are immutable, a component can not be directly referenced after creation or be called methods on, which means we have no concept of an object identity corresponding to the component *from the outside*. This is one of the strengths of the programming model Transmorphic, since mutable state is not part of the picture when rendering is performed. On the other hand *on the inside* a component can actually carry local state that needs to be initialized, updated and possibly cleaned up during the time a component is active. Thus, we *need a mechanism* that allows us to manage this mutable state that is independent of inputs of the user, such as initialization or cleanup routines.

First, Transmorphic assigns the call site of a certain component a global identity such that it can constantly check if future calls (as part of the *immediate rendering mode*) will continue the render the component and/or change the parameters etc. Transmorphic will then constantly monitor that call site and invoke a function from

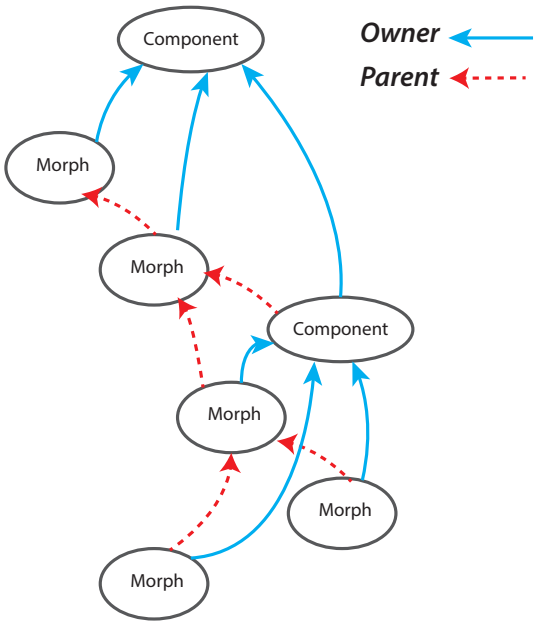

Figure 3.1.: Owner and parent relationships visualized

a set of callbacks that we refer to as the *Component Lifecycle Protocol*. These callbacks essentially denote different transitions a component goes through during its lifetime which includes the following callbacks:

1. *init-state*: Called right before the component is rendered after being absent from the scene (mounting). The return value specifies the initial component local state.
2. *will-mount* and *did-mount*: Called right before and after (respectively) the component was mounted. Mounting refers to a component being called after having previously been absent from the scene. This may be the the very first time a component is called at runtime, but can also be the case, when a conditional continuously decides wether or not a component is being rendered.
3. *will-receive-props*: Called each time a call to the component is about to happen. This callback is especially useful to reason about the changes in properties that are passed to the component in between the render cycles.
4. *will-update* and *did-update*: Called each time the component local state is about or has been updated respectively.
5. *will-unmount*: Called when a component is stopped being called in the render cycle.

This extrapolated finite state machine allows us to define imperative abstractions within the context of an entity that to the outside provides an entirely functional interface, free of mutable state.

Listing 3.10: Lifecycle callbacks all correspond to Clojure protocols that one can choose to implement in order for them to take effect.

```
(defcomponent bob
 IDidUpdate
 (did-update [self props] ...)
 IInitState
 (init-state [self props]
    {:foo 42
     :bar "Hello World"})
 IWillMount
 (will-mount [self props] ...)
 ... )
```

3.2.3. Hand

The last ingredient that is needed to enable interactive interfaces is an abstraction that allows us to model the current user that is interacting with the morphs in the scene. For this, Morphic uses the abstraction of a *hand* to let the mouse cursor interact with the different morphs through either grabbing or dragging behavior. In Transmorphic, the hand is itself represented by a very small rectangle morph which constantly tracks the current position of the cursor. At the same time, the hand is also checking wether or not a morph in the scene graph has been focused by the hand. Hand focus becomes activated once all of the following conditions hold:

1. The left mouse button is pressed.
2. In order to prevent erroneously capturing a simple click we also require that the mouse has meanwhile moved by a certain threshold distance.
3. Lastly, in order to work hand in hand with the actual control flow of the view, the focused morph has to specify in its props that asks for hand focus.

Besides asking for hand focus, the morph should also implement several callbacks in order to influence the progression of the application state accordingly. It is important to understand that in Transmorphic the hand **does not constitute a tool for performing direct manipulation**, but merely exists to trigger morphic specific events that can then influence how the application state progresses. This is due to the rather strict separation between application state and visual representation which requires a different interpretation of hand interactions, in particular with regards to the grabbing and dropping of morphs through the hand. In the following, we will describe how dragging and grabbing behavior differ and how the respective event callbacks can be used to evolve an application state.

Dragging Dragging is the act of manipulating a certain property of a morph by press and hold of the mouse button and then moving the cursor to vary the dragged

value. In the context of Morphic, this mechanism is often employed to alter the position of morphs, or provide interfaces that allow to directly scrub (i.e. vary) values in the application (i.e. scroll bar). The dragging process is divided into three stages, each of which can be intercepted by implementing the respective callback: `:on-drag-start`, `:on-drag` and `:on-drag-end`.

Listing 3.11: Simplified example of a value scrubbing mechanism implemented though the hand's dragging protocol

```
(ellipse
 {:on-drag-start (fn [start-pos]
                   (reset! self {:init-pos start-pos
                                 :current-pos start-pos}))
  :on-drag (fn [delta]
             (let [new-pos (add-points (@self :current-pos) delta)]
               (swap! self assoc :current-pos new-pos)
               (prn "Scrubbed value: " (distance (self :init-pos) new-pos
                   ↪ ))))
  :on-drag-stop (fn [end-pos]
                  (swap! self dissoc :current-pos :init-pos))})
```

The hand automatically keeps track of the currently dragged morph, and supplies all necessary arguments to the callbacks. Initially `:on-drag-start` is called, together with the initial global position of the morph being dragged. At this time, we usually want to perform any necessary state transitions, such as mode changes that need to be undertaken to start the dragging process. After this, for each move that is captured by the runtime, `:on-drag` is called, and passed the delta between the previous and current position of the morph. This information can be used to update arbitrary values inside the model of an application, which makes dragging an ideal basis for the implementation of *scrubbing*[4] of certain values. Once the hand focus is lost, the dragging is stopped and `:on-grab-stop` called where we can perform any necessary cleanup in the application state. Notice that each time the component local state was modified, a rerendering of the morph scene graph was triggered as well. By this, we ensure that the reference to the component local state inside the behavioral properties of the morph always reference the *most recent* version of the state. For instance, `:on-drag` will not be called, before the reset! that happened within `:on-drag-start` has caused the whole scene to be rerendered and the callbacks updated accordingly.

[4]Scrubbing is a widely used user interface paradigm, by which the value of a certain property can be varied through dragging the mouse while the mouse button is being pressed. This allows the user to influence values by directly translating mouse movements to value changes, without the need for separate UI elements, such as buttons or sliders.

Grabbing In the context of the hand in Transmorphic, grabbing is the act of transferring data between two separate control flows of the application. It is very important to understand that hand grabbing is not mutating the structure of the rendered scene, but just an interface for implementing information flow, where the user can drag elements in order to interact with the system. In Transmorphic, the grabbing mechanism can be instrumented to transfer data while also providing a comprehensive visual representation. A grabbing process, consists of two stages, one being the initial grab of a certain morph which is followed by the obligatory drop onto another morph.

In order to implement a grab and drop mechanism in our application, we need to implement the `:on-grab` and `:on-drop` callbacks within the respective morphs of our interface. The `:on-grab` callback must be provided to the morph that represents the grabbed entity for the user and is called with a reference to the hand. The hand provides Clojure's atom interface (`swap!`, `reset!`, `deref`) for storing and reading arbitrary information, meaning the implementation of `:on-grab` can specify what bits of information are to be passed to the hand. By default, once grabbed, a visual deep copy of the grabbed morph will be placed in the hand that can be grabbed around but is discarded as soon as the grab is canceled, or a corresponding `:on-drop` was found and called successfully. Conceptually this behavior is a reasonable middle ground between avoiding mutating the morph scene graph while at the same time, providing a comprehensive visual representation of a grabbed element which is obviously part of a user interface based on a grabbing mechanism.

Listing 3.12: Simplified example of a grab and drop interface. To save space, we have left out the component definitions these morphs are a part of.

```
; component A
(rectangle
 {:id "start"
  :on-grab (fn [hand]
             (swap! hand assoc :countable-morph true))})

; Component B
(rectangle
 {:id "goal"
  :on-drop (fn [hand]
             (when (@hand :countable-morph)
               (swap! self assoc :dropped-morphs inc)))}
 (text {:value (str (self :dropped-morphs))}))
```

When the grabbed morph looses the hand focus, it is dropped on the morph directly beneath the hand morph. In case this morph does not explicitly implement the `:on-drop` method, we check the parent and continue bubbling up the request until the root of the world is reached. If we reached the world root, yet no `:on-drop`

was explicitly implemented, the grabbing process is discarded, which discards the information inside the hand as well as the visual copy of the grabbed morph. In many cases, the user may want to prevent the lookup for `:on-drop` to continue outside the scope of the current application, and can do so, by providing a `:on-drop` method in some wrapping morph that defaults to cancel the dragging process.

3.3. Direct Manipulation in Transmorphic

When introducing the concept of direct manipulation into a system like Transmorphic, we are faced with one essential problem: If there is no mutable state that directly represents the appearance of a system, what then, are we supposed to change in accordance to direct manipulation by the user? Mutating the application state in this situation is often times not sufficient to yield the desired result, since most parts of the morph scene graph may by definition not be influenced by application state at all. Therefore, in Transmorphic, the answer to this problem is to start treating the symbolic description that is responsible for the rendered scene as a mutable entity, and consequently *translate* direct manipulation of a morph scene graph into *code transformations*. We will see that this translation mechanism is very elegant, in that it solves two problems at once: For on we are able to introduce a meaningful implementation of direct manipulation in a functional morphic, by immediate code transformation, compilation and evaluation of the symbolic description. At the same time, the bidirectional relationship between visual representation and symbolic description is established, and the programmer is able to incorporate direct manipulation into the original symbolic description of a component definition.

3.3.1. Halo

While the hand is the main building block to enable interactiveness of morph interfaces, the halo is the primary tool that enables the direct manipulation of properties and structure of the morph scene graph. The visual aspect of Transmorphic's halo interface is directly derived from the one found in Lively Kernel [33] but there are several differences in the actual semantics behind the actions that the halo provides: While the halo in Lively Kernel is mapped to state manipulation, the halo in Transmorphic is a graphical interface to the underlying functional lenses that eventually perform code transformations.

Selection Mechanism The halo selection of a certain entity is achieved by pressing the meta key and performing a left click on that entity. This will summon the halo and display a surrounding rectangle around the entity alongside a set of handles that allow the direct manipulation of the entity. In case the entity is provided the `:id` property, it will display that value below the selected morph or component. In other Morphic based environments, such as Lively Kernel, there is no distinction between morphs and components, and a mapping between visual entities and morphs is completely sufficient. In Transmorphic however, we want the programmer to be able

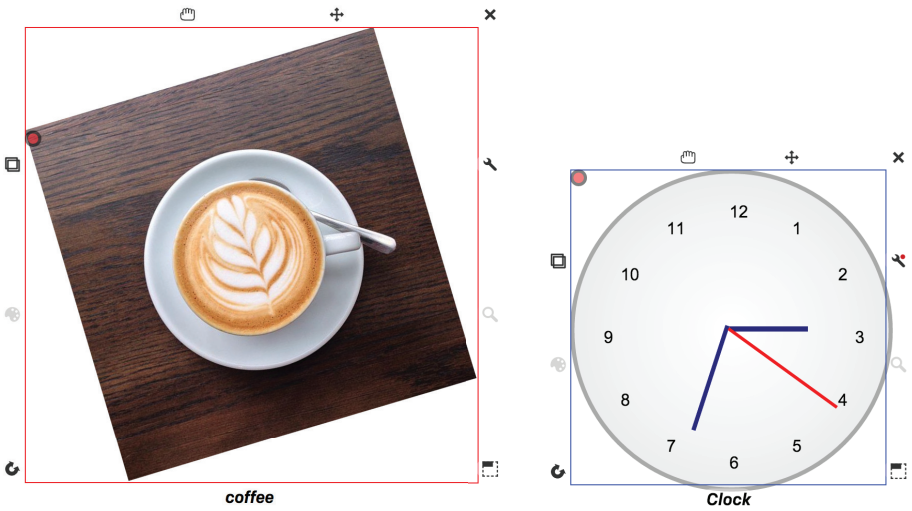

Figure 3.2.: Visual representation of selecting a morph (left, highlighted in red) vs. the selection of a component (right, highlighted in blue)

to evolve morphs as well as components through the halo mechanism, since both directly contribute to the rendered end result. We therefore need to provide a mechanism that makes morphs and components equally accessible from the graphical representation of the application.

When we click on an entity, the halo will always first select the underlying morph, however we are able to reach the corresponding components by propagating the halo selection further up the morph hierarchy. Selection propagation happens when the we repeatedly select the same visual entity in the rendered scene. Usually, the default behavior of the selection propagation is, to pass the halo selection to the *parent* of the already selected entity until the root of the world is reached, in which case the selection starts again from the first morph encountered below the cursor. Transmorphic enhances this mechanism, in that it not only passes the selection to the parent, but first checks if the current entity is the root of a component in which case the corresponding component, namely the entity's *owner* is selected next. Since a component is both assigned a parent as well as an owner, this same pattern continues from there on: In case the selection propagation is triggered again, we check for the component wether or not it is the root of its owner, in which case we propagate the selection to the owner, or else to the parent of the component. To make this slightly different selection mechanism more clear, please take a look at Figure 3.3 where such a propagation is explained in the running example alongside the corresponding symbolic description.

The visual representation by itself, does not convey wether or not we have selected a component or a morph, since components are conceptually just "groups" of morphs. Therefore the halo visually distinguishes between the selection of a morph and the selection of a component, as can be seen in Figure 3.2.

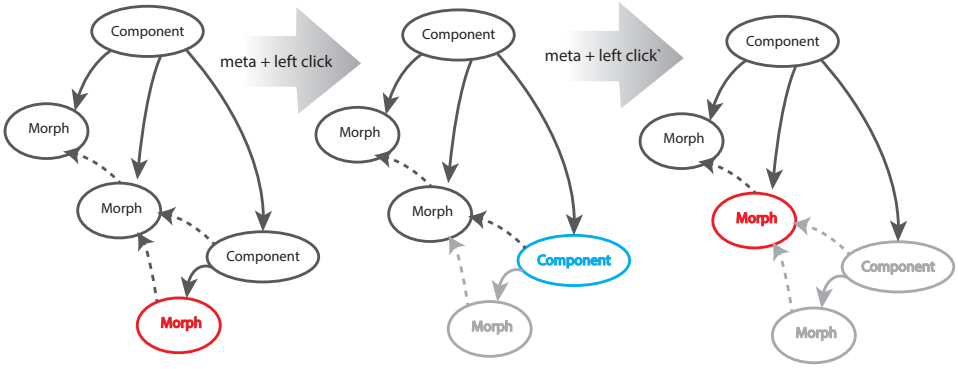

Figure 3.3.: Step wise propagation of the halo selection after repeatedly performing a meta click (alt + left click) on the same target

Altering Properties The most simple type of transformation that the halo provides, is that of the manipulation of properties. The halo carries a set of handles that allows the resizing, scaling, rotating, dragging and stylization of the inspected morph. Essentially this just maps to the setting of `:extent`, `:scale`, `:rotation` or `:position` properties of a morph/component. In addition to that, certain types of morphs may carry properties for which the halo offers specialized interfaces: This includes the text morph's font styles or the polygon morph's vertex array, which can be manipulated by click, drag and drop of vertex markers. In the case of components, the halo does not perform any kind of analysis that ensures that the setting of a certain property actually has the desired result. For example the programmer can cause unexpected behavior, when providing a custom component definition that internally maps the `:extent` property to the `:position` of its root. Conventional property names should therefore not be used to influence the custom behavior of that component, since this effectively undermines the ability to later on operate on this component via direct manipulation.

Grabbing Morphs and Components Next to the mutation of properties, the halo allows to alter the structure of the morph scene graph as well, namely the addition and removal of morphs or components. Again, Transmorphic is able to incorporate these changes, by mapping them to transformations within the symbolic description, i.e. literally adding or removing function calls that are responsible for rendering certain morphs or components. Notice that this also includes the act of copying a certain morph or component which is just the copy and paste of a function call. The process for grabbing and dropping with the halo is very straight forward: Once the grabbing process is initiated, the grabbed morph or component is removed from its parent, and can then be moved by further dragging the halo to a certain position. When grabbing stops, the morph or component is always dropped on the morph (and not the component) that is directly beneath the current position of the cursor. It is however also useful to be able to drop morphs and components on other components, which is possible by holding the shift key while the grabbing is stopped. When the

shift key is pressed throughout the grabbing process, the halo changes the drop target to the owner component of the morph it would have initially been dropped on.

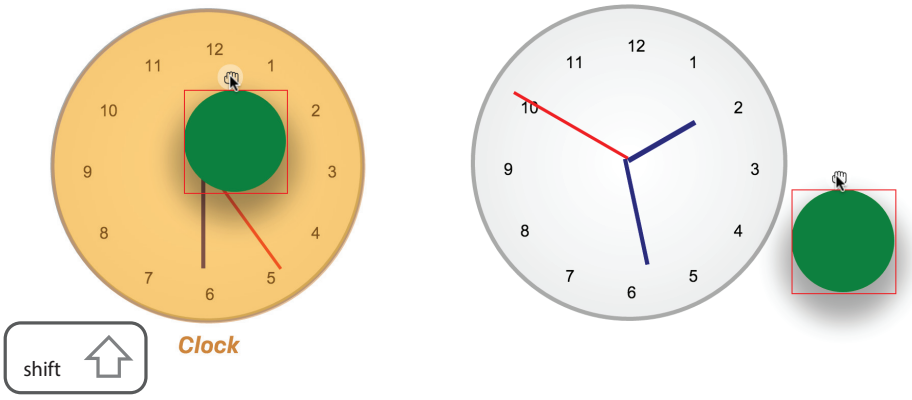

Figure 3.4.: The component that is about to receive the dragged entity (if dropped) is being highlighted by Transmorphic.

Inside this dropping mode, the halo will also highlight the respective component that is about to receive the grabbed target, by highlighting the root of that component as shown in 3.4. Grabbing and dropping slightly differs between morphs and components, in that components will also carry their behavior with them, while morphs will only maintain their visual appearance. If for example a certain control element inside an interface serves a certain role (i.e. a checkbox that toggles a certain value), it will lose that behavior if it is just dragged by itself to a different location. In order to preserve that toggling behavior, the component that the element belongs to needs to be dragged instead, else the behavior is not preserved!

3.3.2. Reconciliation Lens

Internally, Transmorphic makes use of functional lenses to establish a bidirectional mapping between symbolic description and graphical representation. As described, lenses are an abstraction that allows us to extract a view from a given source and then put back an updated view which will cause the source to be updated accordingly. In the following we will give an overview about the behavior that is provided by the lens while the internal API to work with the lens, will be described in the next chapter, where we will describe Transmorphic's current implementation.

We first need to clarify what exactly constitutes the symbolic description and the visual representation, since this is what lenses ultimately translate between. The internal representation of the symbolic description is derived from the set of all namespaces that contain a component definition in the context of the currently run-

ning instance of Transmorphic. Each component definition resides internally as a lisp expression, namely a list structure that can either be updated, evaluated or compiled as needed. In addition to that, Transmorphic further holds a copy of each component's definition for each call context it got used in respectively. The reconciled copy will then serve as a *new* component definition that takes the place of the original component definition at the call site responsible for rendering the modified component, while the unaffected call sites will continue to refer to the original or differently transformed component definition. The other data structure, namely the scene graph, is represented as a tree structure of all the rendered morphs and components. Within this tree, each morph or component carries a globally distinguishable identity such that direct manipulation operations can be applied correctly.

Let us now look at what `GET` and `PUTBACK` correspond to in the context of Transmorphic: When we want to `GET` the view from the symbolic description of the system we just evaluate the current symbolic description of the system together with the application state. We could also say that `GET` is just the process of rendering a component to retrieve the scene graph.

When we perform direct manipulation, we apply a change inside the graphical representation, which Transmorphic does by manipulating the entities in the scene graph. As we will see, these changes only include three types: Changing a property, removing an entity and the addition of an entity.

Finally the `PUTBACK` will take the manipulated scene graph and lookup the symbolic descriptions, which are affected by the manipulations. It will then evaluate to a transformed symbolic description where the direct manipulations are reflected accordingly. The lens ensures that reconciliations are restricted to the respective call site of the component, where the direct manipulation was actually applied to. In 3.5 an example is shown, where a reconciled component definition is created and calls to the old component definition are being replaced up the call graph that used to render the component initially. All the other call sites where the component is also used can still reference the "old" version of the component and remain unaffected. The exact process behind the reconciliation mechanism, will be described in detail in the next chapter. For now it suffices to understand that Transmorphic effectively manages to precisely reflect changes in the scene graph as changes in the source code, responsible for that scene graph.

Transmorphic's reconciliation lens is in fact a *very well behaved* lens, which we can check by analyzing the lens's behavioral aspects:

Acceptable Since `GET` will always provide us with the most recent rendered scene graph, changes in the scene graph are always be reflected accordingly after a `PUTBACK` together with a recompile has happened.

Stability When no changes have been applied to the scene graph, there will be no update in the symbolic description in case of a `PUTBACK`, which also guarantees stability.

Forgetful The reconciliation lens will always completely overwrite the previous symbolic description in Transmorphic in case a `PUTBACK` is performed.

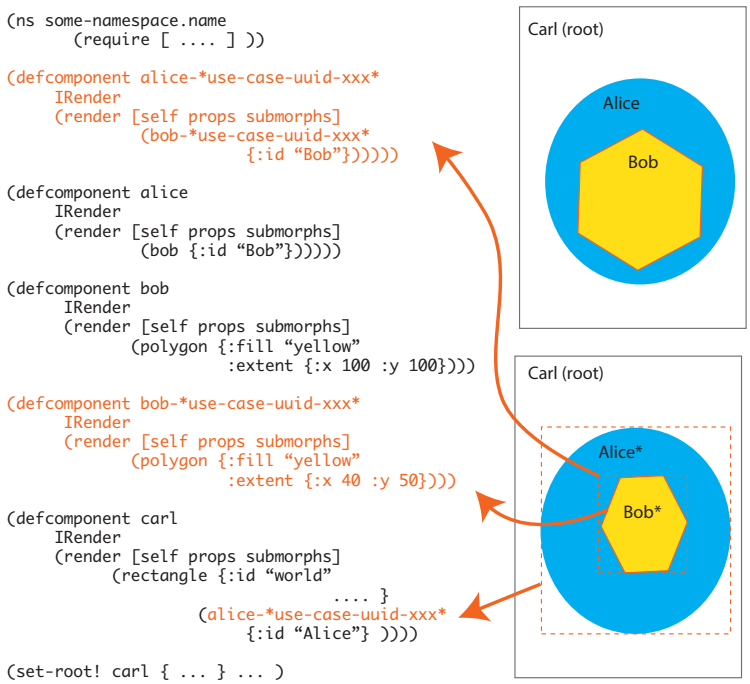

Figure 3.5.: Example of a change to a morph that is reflected within the owner chain of components. In order to isolate a direct manipulation to a single component, all component definitions that are part of the owner chain are replaced by updated versions (i.e. `alice-use-case-uuid-xxx`) that reference the updated components.

Bijective Looking at our concrete example of the rendered morphic scene graph we see that the visual representation reduces application state and symbolic description to a more simplified description: For example, parts of the view code that are currently not "active" are not represented in the view, as is the case for many parts of the application state that are decoupled from the GUI of an application. Therefore Transmorphic's reconciliation lens can not be *bijective* since the extracted views of the source usually discard certain information which is needed to perform an update in the PUTBACK. A successful PUTBACK therefore always requires that the old symbolic description is also provided alongside the updates performed on the entities.

3.3.3. The Role of `:id`

The declarative description of a morph scene graph does not provide a concept of object identity since every entity is immutable and essentially "replaced" each time the application state triggers a change in the visual representation of the application. However to provide an interface for a morph scene graph that can be changed through direct manipulation, Transmorphic needs to be able to uniquely identify every entity inside the scene graph. Since the world an entity resides in, is at any time a well formed tree, we are able to globally address each entity directly if every entity in a tree is distinguishable from its siblings. In Transmorphic this distinction can be mostly done automatic, by combining an entity's position inside the submorph array with the part of the symbolic description that is responsible for their rendering, see Listing 3.13 (that is the actual function call that creates the entity in the first place).

Listing 3.13: Example of a morph composition, where the submorph index is not sufficient to reliably infer an identity, yet combined with the location inside the source code, Transmorphic still manages to differentiate between each morph.

```
1 (rectangle
2  { ... }
3  (if (sunny?)
4     (ellipse {:fill "yellow"
5               :extent { ... }
6               ... })
7     (image {:url "clouds.png"
8             ...})))
```

However, things start to become problematic when the submorphs of a certain entity are *dynamic* and at the same time are *derived from the same symbolic description*, which frequently happens when interfaces are interactive and respond to changes in the data of the application. Imagine for example a map statement that renders a list of tweets with respect to the first 100 most recent entries in the feed of a certain user

as shown in Listing 3.14. In this case `tweet` is a previously defined component that renders a respective tweet given all the relevant data that is required for a tweet.

Listing 3.14: An example for a morph composition, where Transmorphic can no longer infer an identity onto the rendered `tweet` components

```
1 (listmorph
2  { .... }
3  (map #(tweet {:data %}) (fetch-tweets 100)))
```

In this concrete setting Transmorphic is no longer able to infer the identity of each rendered tweet component in case the collection of the 100 most recent tweets changes over time, since neither the index nor the function call in the symbolic description is sufficient to determine the identity of a tweet that was rendered. Also notice that once we loose the identity of a certain component, all the identities of the entities that are yielded by its render method are lost as well.

For these reasons it is mandatory to distinguish siblings from each other, in case a dynamically changing collection of morphs is derived from a single morph or component declaration in the symbolic description. This is done, by providing an `:id` property in addition to the usually passed properties which needs to be unique among all siblings. Internally Transmorphic will then incorporate the value of `:id` into the process of synthesizing the UUID of a morph or component in order to ensure an identity thats independent of the current state of the currently rendered structure of the morph scene graph. If no `:id` property is provided although being necessary, Transmorphic will not prevent evaluation or compilation, however direct manipulation will likely lead to unintended or undesired changes in the visual representation and symbolic description respectively.

Transmorphic's identity inference currently breaks down, as soon as the programmer edits a certain symbolic description by hand and triggers a save and compile of that changed component definition. This is for example the case, when the programmer saves a description at runtime through the *Function Editor*, which we will cover further down this chapter. Since there is not restriction on how the programmer can alter the symbolic description of a certain component, we can also not reliably maintain the identity inference. For example a morph with the exact same `:id` value can, after the change in the source, be assigned a completely different role, with different attributes, behavioral properties or even different morph type. Also it can be confusing the programmer to enter changes in the symbolic domain, which all of a sudden no longer take the desired effect once rendered. For this reason, once Transmorphic is notified that a certain component was redefined and recompiled, all changes that are associated to morphs or components yielded by the redefined component, are discarded.

3.3.4. Orphanization

One key problem we are faced with, when we perform code transformation to reflect direct manipulation changes, is related to the bindings inside a symbolic expression. Imagine the process of copying and/or grabbing a morph/component and then placing it into a completely different place inside the world. In these cases, where a morph or component is removed from the the lexical scope it was initially declared in, we need to consider what happens with properties or submorphs that are bound to other variables inside that function. For example callbacks to mouse events, contain variables that are bound in the functional scope of the previous owner's render method, and can most likely not be resolved reliably when the morph or component is later added to a different scope. This is problematic, since that would mean that we could not realize the removing, copying or adding of morphs which hold properties bound to their current functional scope, since we could not successfully recompile such a description. For this reason, Transmorphic by default applies a process called *orphanization* to any morph or component that is removed from its current owner. Orphanization means that the morph or component is detached from the functional scope it was initially declared in by replacing all behavioral properties with empty functions and all relative and/or primitive properties with the value they currently evaluate to. This effectively clears the entity off any symbolic expressions that are bound to the old scope, such that it can be seamlessly integrated into any new functional scope it may be added to in the future.

Listing 3.15: Example of an orphanized ellipse morph that was removed from a previous symbolic `let`-expression, where several properties where defined through variables

```
(let [a 42
      b "foo"]
     (ellipse {:extent {:x a :b a}
               :fill "green"
               :on-mouse-down (fn [_]
                                (prn b))}
              (let [c "bar"]
                  (text {:value (str a b)})))))

; Will be orphanized to:

(ellipse {:extent {:x 42 :y 42}
          :fill "green"
          :on-mouse-down (fn [e])}
        (text {:value "foobar"}))
```

3.3.5. General Abstraction Preservation

In cases where we detach entities from their original context, and place them into an entirely new one, it is reasonable to apply orphanization, since the *role* of the entity can not be inferred automatically. However if the entity is not removed, yet we manipulate its properties or submorphs, it becomes crucial to preserve the symbolic expression the entity is embedded in, together with any variables that define the entity's properties. Symbolic statements such as loops, binding expressions or anonymous functions constitute vital parts in what defines the *role* of a certain morph or component inside the interface, and we need them to be preserved as well in order preserve the behavior of the GUI in the presence of direct manipulation. Transmorphic therefore modifies the corresponding symbolic description of a morph or component, without disturbing the "symbolic context" it is embedded in. To illustrate this property more clearly, let us for a moment look at Listing 3.16 to understand the goal we are after.

Listing 3.16: Example of an ellipse morph that is wrapped inside a `loop` expression, yielding a collection of separately placed ellipses, alternating in color

```
(loop [i (range 10)]
   (ellipse {:id (str i)
             :position {:x (* 10 i) :y (* 10 i)}
             :extent {:x 42 :y 42}
             :fill (if (even? i) "orange" "dodgerblue")
             :border-color "brown"}))
```

Let us assume that an edit session is up and running and one of the morphs that is yielded by the loop statement in the example is also selected with a halo. If the programmer now starts to manipulate the morph through the halo, we can see that Transmorphic will preserve the symbolic description as much as possible, thereby only adapting the property that was changed by the halo in Listing 3.17. Only the changed properties will replace the previously existing property value (`:extent`), or symbolic expression that the property was derived by (`:fill`) in the symbolic description.

Once we continue with performing changes in the graphical representation, we also see that changes to the structure are directly applied to the the loop statement, again without removing the abstraction, see Listing 3.18.

Since the eventual *behavior* of the direct manipulation is directly influenced by the way we reconcile changes with our symbolic description, it is worthwhile looking at what actual behavior the reconciled code exhibits. Notice that not only the adapted function call influences the end result, but also the symbolic expression (in this case the `loop`) that a morph or component may be embedded in. For example in Listing 3.17 the new value for `:extent` and `:fill` will affect *all* of the ellipse morphs

Listing 3.17: Reconciled symbolic description of Listing 3.16 after fill and extent have been altered through the halo

```
(loop [i (range 10)]
  (ellipse {:id (str i)
            :position {:x (* 10 i) :y (* 10 i)}
            :extent {:x 102 :y 98}
            :fill "#32CD32"
            :border-color "brown"}))
```

Listing 3.18: Reconciled symbolic description from Listing 3.17 after a rectangle morph has been dropped on one of the ellipses

```
(loop [i (range 10)]
  (ellipse {:id (str i)
            :position {:x (* 10 i) :y (* 10 i)}
            :extent {:x 102 :y 98}
            :fill "#32CD32"
            :border-color "brown"}
           (rectangle
            {:position {:x -5, :y -14.5},
             :extent {:x 47, :y 54},
             :border-color "orange",
             :fill "gold"})))
```

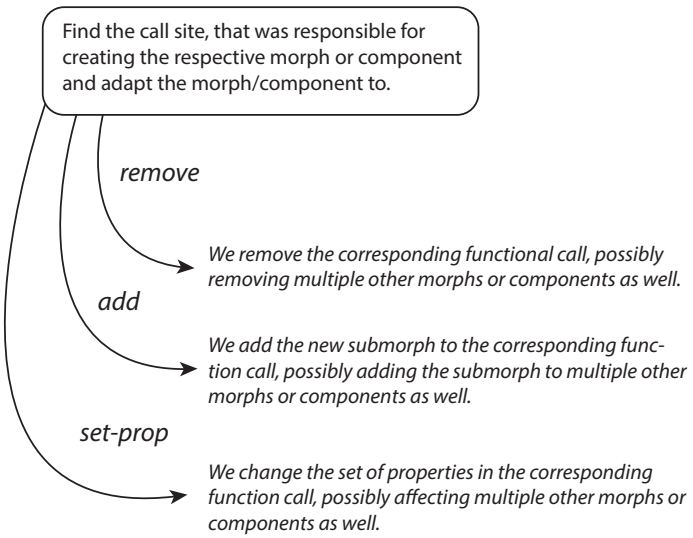

Figure 3.6.: The decision policy for reconciling direct manipulations declaratively

that are derived from this very function call, and not only the single morph we applied the change to. This will causes the manipulation that was initially applied to a single morph to be propagated among all other morphs of the same collection.

The effects by this style of reconciliation are rather different to the common interpretation of direct manipulation, where a change is isolated to the object we are directly operating on. We refer to this reconciliation approach as the *declarative reconciliation*, since it leverages the declarative description of collections to propagate changes to multiple entities at once. While we think that this interpretation of direct manipulation is not suitable to be the default, we none the less decided to include it into Transmorphic, since we encountered various scenarios where simultaneously changing multiple entities can be very useful. Figure 3.6 illustrates the reconciliation strategy again, covering all cases that may be encountered in Transmorphic.

3.3.6. To Isolate or not to Isolate

While declarative reconciliation has its merits, we argue that this should not be the default reconciliation approach, since the propagation of changes can often lead to unexpected results, especially when just presented with the visual representation of the system. It is therefore worthwhile looking for another transformation policy that *always isolates* each change to the very entity it was performed upon. In order to isolate changes within a collection, we need a property that allows us to differentiate between different entities that are yielded by the same function call. This is exactly what the value of `:id` provides us with, so we are able to isolate changes by dispatching the property value on the entity's `:id`. Let us revisit the example from previously where reconciliation happened declaratively and look at how the isolated reconciliation looks like in comparison in Listing 3.19.

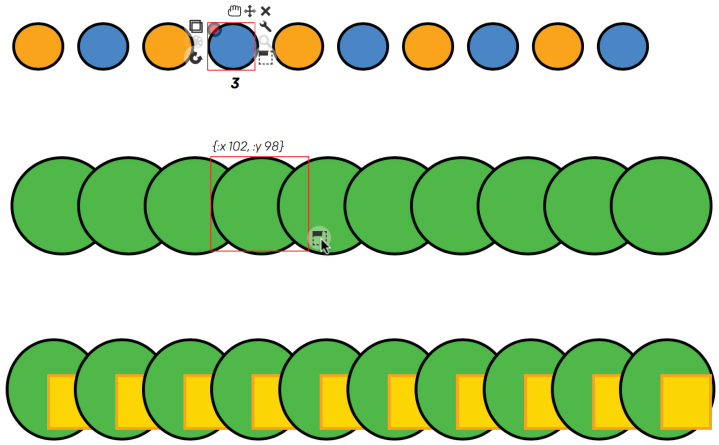

Figure 3.7.: Manipulating an entity that is part of a declaratively defined collection, will propagate the change among all entities that are part of that collection. This picture shows the graphical representations of the transformations in Listings 3.16, 3.17 and 3.18 respectively.

Listing 3.19: Example from Listing 3.16, yet this time reconciled by means of direct manipulation reconciliation

```
(loop [i (range 10)]
  (ellipse {:id (str i)
            :position {:x (* 10 i) :y (* 10 i)}
            :extent (case (str i)
                      "3" {:x 102 :y 98}
                      {:x 42 :y 42})
            :fill (case (str i)
                    "3" "#32CD32"
                    (if (even? i) "orange" "dodgerblue"))
            :border-color "brown"}
           (when (= "3" (str i))
             (rectangle
              {:position {:x -5, :y -14.5},
              :extent {:x 47, :y 54},
              :border-color "orange",
              :fill "gold"}))))
```

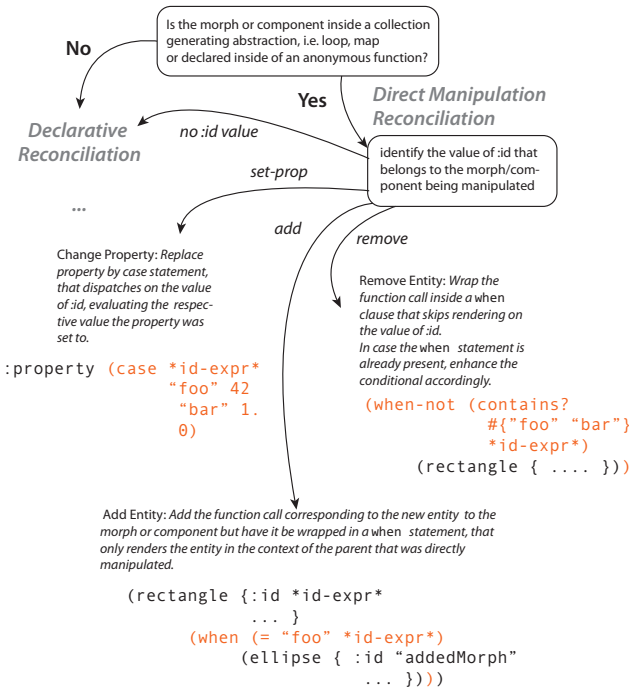

Figure 3.8.: The policy for reconciling direct manipulations in an *isolated* fashion. Here `*id-expr*` denotes the symbolic expression that evaluates to the respective `:id` of the morph or component.

Notice how, instead of completely overwriting the previous property values, a `case` statement is introduced which allows us to assign the property value to the entity carrying the respective `:id`. Also the rectangle that was previously added implicitly to all of the ellipse morph's from the loop, is now just added to the directly manipulated morph. This transformation strategy is in fact rather *imperative* in nature, in that we introduced a conditional for each property or morph that was manipulated. We refer to this kind of reconciliation as *direct manipulation reconciliation*, and we outlined the corresponding decision network in Figure 3.8.

3.4. Function Editor

The Function Editor is the core building block in Transmorphic for bridging the gap between symbolic description, and graphical representation of the morph scene graph. The idea behind the Function Editor is closely related to the one of the Object Editor in Lively Kernel, where the programmer is able to directly change the state and behavior that resides on a certain morph. As the name already implies, it is however not mutable state encapsulated by objects that are modified by the Function Editor, but abstractions, namely the definition of components, instead. In the context of Transmorphic, this means that we are able to change or create the definition of

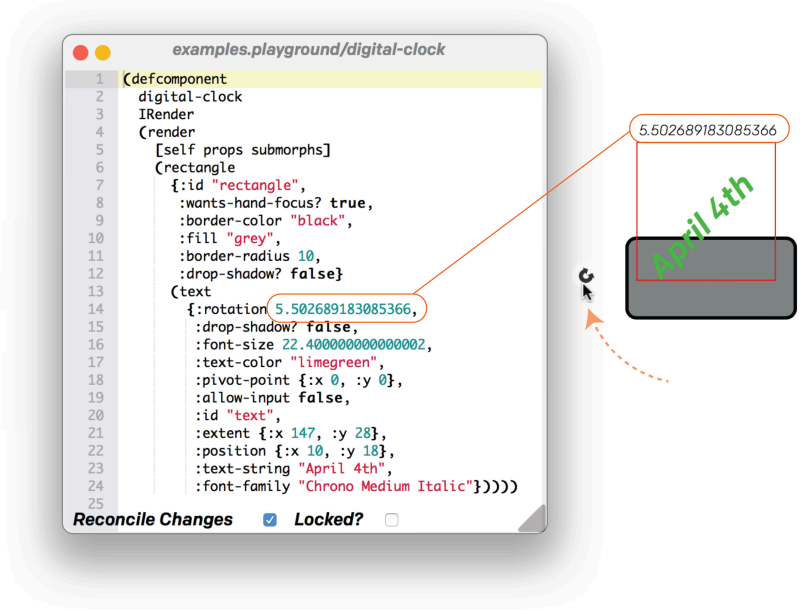

Figure 3.9.: Function Editor in action: The direct manipulation of the text morph's rotation property via the halo (right hand side) is complemented by the Function Editor transforming the component definition accordingly (left).

a Component by either altering the symbolic description directly, or through direct manipulation of the scene graph in the graphical representation. It is important to understand that the function editor changes the symbolic description within the *original source files* of the project and not the symbolic description that is kept internally by Transmorphic. The function editor does however allow us to incorporate the changes applied to the internal symbolic description to the definitions inside the editor such that the programmer is able to evolve the source code by both, manual editing and direct manipulation.

3.4.1. Editing Morphs

Depending on wether the halo selected a morph or a component, the function editor will start the edit session in a different mode. In case the halo selected a morph and we decide to open the Function Editor, Transmorphic creates a new definition of a component that contains the symbolic description of the selected morph (and all its submorphs). Before the edit session is initiated, the programmer is prompted to select, or define the namespace that the new definition is supposed to reside in. This is done, by a drop down menu that automatically expands from the halo's edit button, where the user can either select an existing namespace, or declare an entirely new one. This ad hoc generated symbolic description does not include any kind of abstractions (i.e. let or map statements) besides calls to morphs or components. In fact it is the

orphanized version of the morph that was previously selected by the halo. The idea behind this behavior of Transmorphic is that the programmer is able to create new abstractions (components) by starting to edit a selected morph in the graphical representation of the morph scene graph. For example, the programmer may start by assembling different visual parts (morphs and components) with each other, then select the one that is supposed to function as the root of the new component, and from there on create and evolve a newly defined component. When the initial compilation is initiated, Transmorphic replaces the morph from which the new component was derived, with a call to the newly defined component. By default, this call to the new component is parametrized with the same props as the morph that used to reside at its place beforehand. After this compile and swap transactions, the function editor automatically transitions into the component editing mode, which we will describe in the next section.

3.4.2. Editing Components

The second mode of the function editor is triggered when already existing components are being edited. This mode is reached by either starting the function editor when a component is selected via the halo, or once a morph has been replaced by a newly defined component in which case the function editor transitions from morph editing to component editing. When editing the component, the function editor immediately presents the programmer the source location, where the component's definition resides and allows the programmer to alter the symbolic description or trigger a recompilation after a change has been committed. By default the function editor presents the component definition in context of the whole definition of the namespace, allowing the programmer to also add new function definitions to the namespace that may be used as part of the behavioral properties of the morph inside the component's render method. In case multiple different component edit sessions are opened in parallel (different components residing in the same namespace), a change once saved in one editor, is automatically propagated to all other editors in the same namespace. This mechanism prevents the problem of lost updates and ensures that each editor has a consistent view onto the source of the application when an edit session is active. The programmer may apply almost any arbitrary alterations to the whole namespace definition such as declaration of global variables, or additional dependencies for the namespace. Currently this freedom in editing the symbolic description is restricted to one aspect, namely the renaming of components. Because the mapping between rendered entity and symbolic description is entirely name based, renaming a component or namespace out of spite either breaks the reconciliation of the source, or the whole compilation in case required dependencies in other namespaces can no longer be resolved. For this reason, renaming of components should be performed through a separate menu, which allows Transmorphic to react to the renaming, and perform the necessary changes to source and runtime in order to let the current edit session evolve undisturbed.

3.4.3. Incorporating the Reconciled Source

As mentioned, behind the scenes Transmorphic already applies code transformation as a means to realize the desired changes in the graphical representation of the system, yet all of this was hidden from the programmer up to this point. In order to let Transmorphic incorporate the changes due to direct manipulation directly into the source within the function editor, the "Reconcile Changes" box in the bottom of the editor window has to be checked. If reconciliation is active, the editing of the symbolic description is temporarily turned off, and is instead replaced by the symbolic description that is being generated by Transmorphic at runtime to apply the direct manipulation changes. One example of this behavior can be seen in Figure 3.9.

The "Reconcile Changes" box may be toggled any time during the edit session of a component, however this will also toggle between two different alterations of the symbolic description: One containing the changes done by the user through editing the source, and the other that was created by incorporating the changes in the symbolic domain. As of now, Transmorphic does not provide a mechanism that allows to also combine incorporated changes and edit changes simultaneously, since the source reconciliation always assumes a static structure of the symbolic description that it is able to reason about. It is also important to understand that Transmorphic restricts the incorporation of reconciled source code to the scope of the component that is currently being edited and explicitly excludes the updated calls to automatically generated component definitions. For example in Figure 3.5, where for a modified use case the `PUTBACK` created a custom component definition for that manipulated component, the call to the custom component definition *is not* part of what is included in the code managed by the function editor. Since the use case related component definitions serve a solely technical reason, including them in the source code reconciliation of the function editor would rather lead to confusing the programmer, instead of serving a meaningful purpose. The function editor does however preserve the knowledge about these "altered" components, and notify the programmer about changes that are "hidden" behind component calls and not reflected in the source, in order to prevent the accidental loss of direct manipulation changes.

4. Implementation of Transmorphic

Setup Each Transmorphic project is running as a client session inside the browser, however we currently require a JVM that handles compilation requests, since Clojurescript does not yet support self hosted compilation to a satisfying extent.

Component Local State This is where Transmorphic stores application state that can directly trigger the re-rendering of a certain component. Of course the actual state that influences the final application may also reside in global variables that code can later reference, however this information is not part of the bookkeeping of Transmorphic.

Morph Scene Graph The data structure that reifies the current instance of the morph scene graph, taking into account both, what the component's render method returned and what the user introduced as changes through the halo interface. It can be directly passed to the virtual DOM and be rendered inside the browser session in order to be displayed. We make use of the React.js [11] library to render the scene graph into the browser's DOM.

Symbolic Description Transmorphic also keeps track internally, what the current symbolic descriptions of components and the namespaces they reside in are. Note that this is derived, yet separate from the actual source files that belong to the running Transmorphic project.

Direct Manipulation API Transmorphic provides a fixed API for applying direct manipulation operations to the scene graph, which allows us to precisely reason about the changes to the scene graph in order to perform the necessary code transformations when a `PUTBACK` is triggered.

Lenses At the core of the rendering process are Transmorphic's functional lenses, that on the one hand can render the current scene graph out of symbolic description with respect to the component local state (`GET`) and also reconcile the symbolic description with updates that have been performed on the scene graph through direct manipulation (`PUTBACK`).

Internal Optimizations In order to provide a more responsive user interface, Transmorphic employs a set of internal optimization strategies which includes skipping recompilation of symbolic descriptions and only partially rendering the morph scene graph.

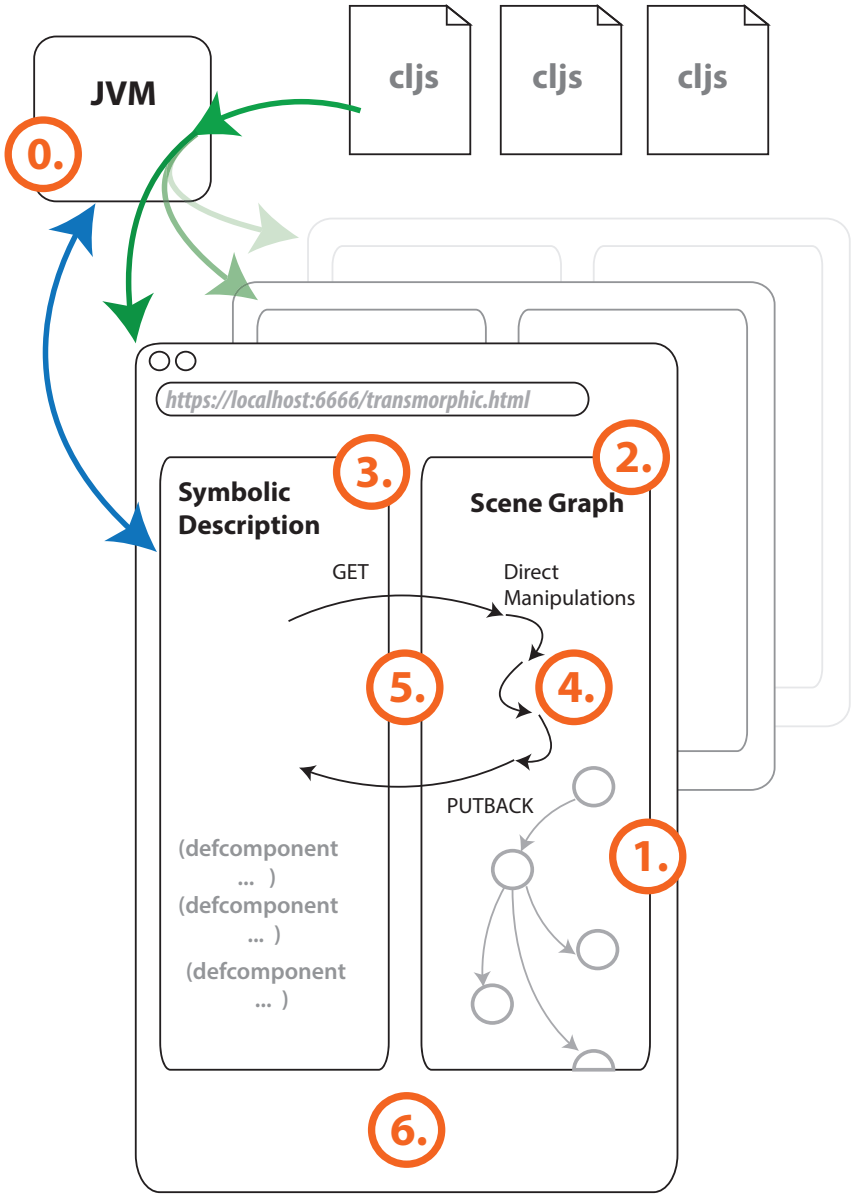

Figure 4.1.: Overview of the different parts that make up Transmorphic's implementation

4.1. Setup

Transmorphic development sessions always run in the context of the browser, which means that the internal data structures such as symbolic descriptions or scene graph are all kept inside the client. Despite Clojurescript being able to bootstrap itself, there currently are wide range of Clojurescript libraries that still require to on be compiled in the context of the JVM. For this reason the current setup of Transmorphic also keeps a JVM running next to the browser session that we use to support compilation in Transmorphic.

Inside the client, Transmorphic stores the scene graph together with the symbolic description within the atom that was provided to each of the `set-root!` function calls, which we refer to as the *world atom*. Each of these world atoms is constantly being monitored by Transmorphic to trigger a render pass once a change mutates the information inside the atom. This includes changes to the symbolic description, changes to the scene graph and changes to component local state.

Compilation can be triggered either by editing one of the project's source files or by recompiling parts of the internal symbolic description. In order to react to changes in the source files Transmorphic automatically computes the smallest set of files that need to be recompiled and then pushes the recompiled artifacts to all connected clients. Symbolic descriptions are updated inside *all* of the evolving worlds within the client, which means that we break isolation in the case that the initial source of the project got modified.

For compilation requests that come from *inside* of Transmorphic, for example in reaction to direct manipulation, we allow the respective world to communicate with the JVM through a socket API. Here we are able to issue compilations of small incremental updates such as a redefined single component definition. This allows us isolate changes in the symbolic description to the respective world and also obfuscates the need to modify and recompile the original source files.

It is important to note that all of the structures and functions we are about to describe in the following are implicitly bound to a certain world context, meaning they can not be used across different evolving worlds. In case we want applications in different worlds to talk with each other, we need to implement separate mechanisms that handle the communication between those worlds.

4.2. Scene Graph Representation

The scene graph is an object-oriented representation of the rendered morph scene, which is retrieved by evaluating the render method of a certain component. During the render process, a global identifier is assigned to each morph and each component that is part of the rendered scene, from which a flattened tree can then be derived such that we can store the tree inside a key value store. To get an overview of the information that is kept inside the scene graph, take a look at the schema at Figure 4.2.

This key value store is essentially a database that maps the morph or component's UUID to the respective morph or component entry which contains all the necessary

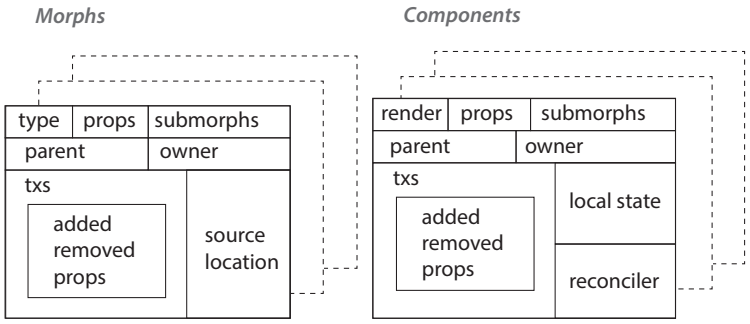

Figure 4.2.: Schematic overview of Transmorphic's internal representation of the rendered scene graph. Components and morphs are stored separately since they store different kinds of information.

information for Transmorphic to either render the scene to the DOM or react to direct manipulation. First are the extensional attributes that are directly used to generate a visual representation of the scene graph which includes passed *properties (props)* and the array of *submorphs* for a certain morph or component.

In order to traverse the scene graph efficiently, we are provided with the already mentioned submorphs but in addition to that also the *owner* and *parent* reference, each storing the UUID of the respective owner component or parent entity.

Besides being the data structure that can be directly passed to the virtual DOM to be rendered, the scene graph also stores the component *local state* within the entry of each component, since this is also something that is directly dependent on the inferred identity of a component. The dictionary that is denoted by *txs* stores all of the performed direct manipulations, which will be described in more detail in the end of this chapter. In practice, the scene graph stored in a certain world atom is always derived from the component that was defined to be the root of the world. However theoretically we can retrieve the scene graph based on any defined component, given that we have its local state and a set of properties we can pass to it.

4.2.1. Identity Inference

We already mentioned that in case of ambiguity between the identity of rendered morphs or components, the programmer is required to provide the :id property, to uniquely differentiate siblings from each other. However, for Transmorphic to be able to mutate the rendered visual morphs after they are rendered, we need to be able to address them by a reference that identifies them uniquely in the context of the *whole* scene graph. It is important to understand that the concept of a "reference" is not available by default in our system but instead needs to be re-introduced. When defining a component in Transmorphic, any code that in the end yields a visual element, is purely declarative. When a morph is supposed to be rendered in the scope of a component, it is done by performing a function call and passing the required arguments which creates an immutable value that can not be manipulated throughout the rest of the render pass. This restriction is intentional, and is one of the

core reasons, why declarative code is easier to maintain and reason about, since there is a single direction in which information flows in order to yield the rendered scene graph. We loose identity of the rendered elements, but gain a better understanding of what we actually render in the end.

Since within the symbolic description of a morph scene graph, there is no identity, Transmorphic applies a strategy that assigns a universal identifier to each rendered morph and every component *after* it has been rendered, so that it can be uniquely identified among all other rendered elements inside the current world. This identifier is automatically derived from the absolute path to that entity inside the world, and the respective local `:id` properties of the morphs or components that are encountered throughout the traversal of that path. Once the component or morph has been rendered and assigned the corresponding identifier, it is placed inside a key value store that maps identifier to the morph or component entity. Ambiguity, meaning there are branches which we can no longer distinguish from each other (as described in Section 3.3.3 where we indicated the importance of `:id`), will cause Transmorphic to loose the ability to infer identifiers to all the rendered descendants further down that branch. Consequences of identity loss are various, ranging from the inability to correctly introduce changes through the halo up to unintended convolution of local state of different components and should therefore be avoided at any cost.

4.2.2. Component Submorphs

There is one aspect about Transmorphic's scene graph, which at first creates ambiguity with regards to the parent relationship. As mentioned, we can pass an arbitrary number of morphs or components as submorphs to a component which itself is completely free to place the submorphs anywhere inside its rendered morph hierarchy. For example the component may decide to place its submorphs immediately as the submorphs of its root, but may also place the submorphs somewhere else, deeply nested inside the hierarchy of morphs it is the owner of (see Listing 4.1).

Listing 4.1: Example of the submorphs being rendered further down the morph composition encapsulated by a component

```
(defcomponent morph-wrapper
 IRender
 (render [self props submorphs]
       (rectangle
        {:id "HourPointer"
         ... }
         (ellipse {:id "wrapper"
                   ... }
                  submorphs))))
```

Listing 4.2: Example of a component that repeatedly renders its submorphs in different places

```
(defcomponent repeater
  IRender
  (render [self {:keys [times]} submorphs]
        (map
          #(rectangle {:id "wrapper"
                       :position %}
                      submorphs)
        (get-wrapper-positions times))))
```

We could even imagine components that render the submorphs not a single but multiple times, functioning more like a repeater than a simple wrapper of the morphs that are passed to them (see Listing 4.2).

It now becomes difficult to say what we can consider the parent of a certain morph that has been passed as a submorph of a component. For example in the extreme case where we repeatedly render the passed submorphs, there is no longer a way to denote a single parent for a passed submorph. We again encounter a conflict between the functional, declarative interface of Transmorphic, and its simultaneous notion of morphs having an identity: While the parent relationship of morphs requires the notion of object identity, the way we issue the render of a morph or component is purely declarative, without the notion of directly referencing the morph or component as a mutable entity. To guarantee that any morph in the rendered hierarchy can still be uniquely identified, submorphs that have been passed to a component are split up into two different entities, one being the *prototype* and the other being the *derived* morph. A prototype is never rendered directly but only receives a visual representation by one of its derivations, which denotes the morphs that are passed to a certain parent morph in the render method of the component. In Figure 4.3 the relationship between prototypes and derived morphs is illustrated visually.

Having established the prototype-derivation relationship, we are again able to uniquely assign the appropriate parents to each morph. Prototype morphs always have the component as the parent, they have been passed to, while derived morphs are always the children of the morph or component they have been passed to in the scope of the render method. Notice that the owner of both prototype and derived morph is the same, namely the component from which they initially received their properties.

4.3. Representation of the Symbolic Description

Transmorphic keeps an internal representation of the running project's symbolic description that is separate from the source files. We determined that a separation between project source files and the symbolic description at runtime, is useful for

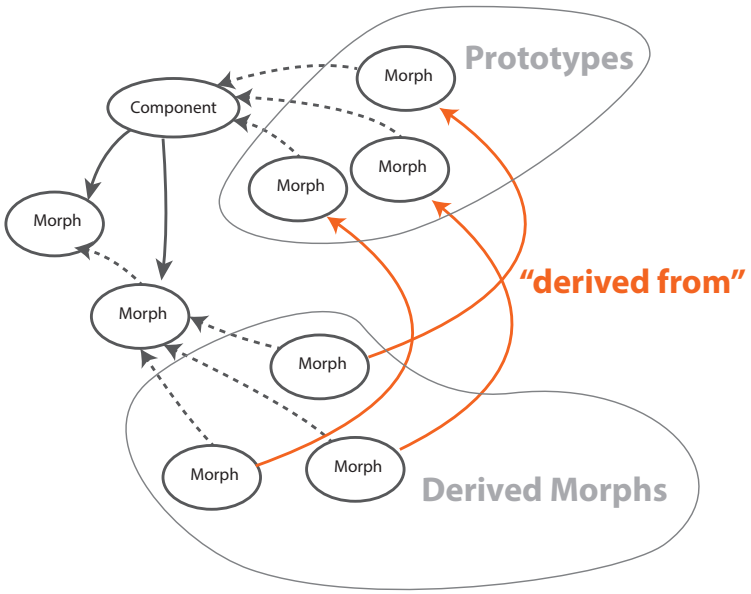

Figure 4.3.: Relationship between prototypes and their derived morphs

multiple reasons: For one, the programmer should still have the final word, when it comes to modifying the source code of the application: While source transformation is a means to implement the direct manipulation, the runtime should not interfere with the original source files and perform source authoring on behalf of the programmer. Encapsulating the source transformations in a separate structure allows us to flexibly create symbolic descriptions internally that reflect the changes due to direct manipulation without directly affecting the sources. At the same time we are also able to export the transformations to the actual files in case explicitly demanded by the programmer, for example by enabling reconciliation in the function editor. Furthermore, a separate structure in which the symbolic description is stored, simplifies managing the isolation of changes to the description. For example, we can start differentiating between the uses cases that a component appears in by storing separate copies of the symbolic description and only apply source transformations to the specific use case. Likewise, the compilation of different components can also happen in an isolated fashion and we are not required to recompile entire namespaces in cases just a single component definition changes.

We first store all of the different namespaces together with references to all of the components that are defined within them alongside the path to the source file that the namespace originated from. In addition, we also store the compilation context that is used in cases where some of the components need to be recompiled in isolation. The compilation context is initially created when the entire project with all its namespaces is compiled, and is then kept internally to be reused and updated at runtime, when new compilation requests are scheduled. What follows is the list of all defined components each of which is split up into different use-case scenarios. By default, a component only has a single use case that applies to all of the instances

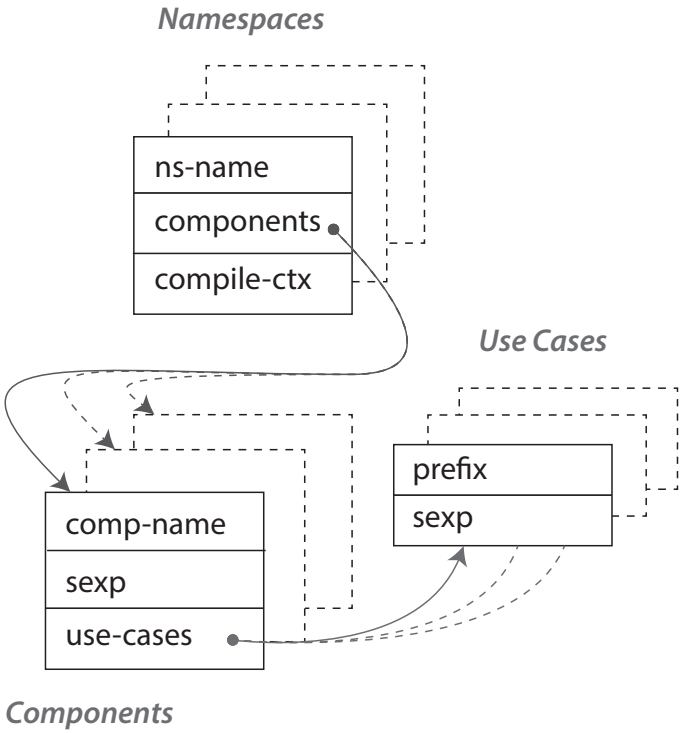

Figure 4.4.: Schematic description of the internally stored symbolic descriptions

it is used in, however every time a component definition is updated with regards to a direct manipulation, a new use case is introduced in order to isolate the change to the instance at hand.

4.4. Direct Manipulation API

In the previous chapter we have seen various kinds of direct manipulations that can be applied to the rendered morph scene graph. By now, we have already learned that the UUID that is assigned to each morph and component allows us to, in theory, precisely define various kinds of changes to the rendered entities. We will now present the internal API for traversing and mutating the morph scene graph in Transmorphic. Notice that the API for direct manipulation is implicitly bound to the context of a certain world, which is why we do not pass the world explicitly as an additional parameter. It should also be noted that the features presented here should be restricted to tooling and other applications that support the development process of the programmer. In no case should this API be used *within* the scope of a component's render method, since various parts of the API are highly stateful and directly interfere with the rendering process. In the following we will use the term "entity" to refer to both morphs and components alike, since many parts of the API apply to both of them in the same way.

4.4.1. Walking the Scene Graph

We will first explain the part of Transmorphic's direct manipulation API that allows to traverse the morph scene graph, which includes the following functions:

```
$morph, $component, $submorphs, $parent, $owner, $props
```

The functions `$submorphs, $parent, $owner, $props` all work with respect to a certain entity (that is a UUID), so we need some way to enter the scene graph at an initial point similar to a handle from where we can start traversing the scene graph. This is what the functions `$morph` and `$component` provide us with: In both cases, a certain morph can be addressed based on its `:id` property:

```
($morph "morph-id")
```

The function will search the main morph scene graph (that is the *world* it is bound to) beginning from the root and return a reference to the first morph that carries the property `:id` of value "morph-id". Since the value of `:id` is often only unique among the siblings of a component or morph, we can pass further arguments to `$morph` or `$component`, in order to specify that a certain value of `:id` has to occur in the chain of parents, like so:

```
($morph "parent-id" "other-parent-id" ... "morph-id")
```

This allows to quickly reduce the number of possible matches, in case we need to hand pick a certain morph and want to perform direct manipulations in a purely programmatic way. Given that have get a hand on a specific morph, we can continue with reading its properties or fetching its parent or the owner component respectively:

```
(let [ref ($morph "parent-id" "other-parent-id" ... "morph-id")]
    ($owner ref)
    ($props ref)
    ($parent ref)
    ...)
```

We can continue traversing the scene graph in arbitrary ways, just we would in a common imperative environment.

```
(let [ref ($morph "parent-id" "other-parent-id" ... "morph-id")]
    (-> ref $parent $parent $owner ...)
    ...)
```

Properties returned by `$props` are fully evaluated, and not in the symbolic form they were initially defined in.

4.4.2. Manipulating the Scene Graph

Transmorphic provides a fixed set of possible mutations in order to reason about the changes in a way that can later be reconciled with the symbolic description. In particular the following set of functions are provided each in two flavors, one for morphs and one for components respectively:

1. `(add! parent-ref ref)` Takes a reference (UUID) of the parent entity and reference (UUID) to another entity that will be added to the parent.
2. `(remove! ref)` Removes the entity referenced by `ref` from the scene graph. Notice, that the UUID entry is still kept inside the key value store and the removed entity can be inserted into the scene graph at any point in time until recompilation may invalidate the entity. This call also causes the orphanization of the respective entity such that it can be added to other contexts afterwards.
3. `(set-prop! ref prop value)` Sets the property `prop` to the value `val` at the entity pointed to by the UUID `ref`. Notice that value may also be a relative property, and not necessarily a primitive value.
4. `(copy! ref)` Allows the programmer to create new components/morphs based on already existing ones. This allows to introduce new elements into the scene graph, without the need to change the symbolic description and recompile a particular component definition. The copied entity is orphanized.
5. `(move! ref new-parent-ref)` Is just a shorthand for a consecutive `remove!` and `add!` call. Just like removed or copied entities, a moved entity is getting orphanized implicitly.

Notice that each of these functions has side effects, in that it causes the scene graph that is stored inside the world atom to be updated. Consequently the names of the functions all carry a trailing exclamation mark, which is the convention in Clojure to denote functions that perform side effects on their behalf. We are able to retrieve a reconciled symbolic description of the affected components, through the reconciliation lenses we will talk about in detail next.

4.5. Source Transformation Lenses

Internally Transmorphic implements the bidirectional mapping between symbolic description and graphical representation by means of functional lenses. We will now see how the different lenses are represented concretely in Transmorphic, how they are being used and how they work internally.

Currently there exist two different types of lenses that are actively part of Transmorphic's reconciliation machinery: The *Direct Manipulation Lens* reconciles the symbolic description and isolates changes through introduction of conditional statements to single morph instances, in case they are part of a morph collection. On the

other hand the *Declarative Manipulation Lens* refrains from introducing conditionals, thereby leveraging the declarative expressiveness of a symbolic expression which in the context of morph collections lifts direct manipulations to apply all morphs or components that were derived from the same symbolic description.

We saw in the previous chapter, how the lenses behave with regards to their `GET` and `PUTBACK` functions. Internally, the lenses can be accessed through two respective functions that take the scene graph representation and return as a result a new symbolic description representation:

1. `(reconcile-declarative symbolic-description scene-graph)`
2. `(reconcile-direct symbolic-description scene-graph)`

Note that these two functions are pure, and only expose the mapping mechanism the reconciliation lenses provide. In order for reconciliation to actually affect the world, Transmorphic provides two side effecting functions, which similar to the direct manipulation operations, are bound to a certain world so we only need to pass a reference to the component that is meant to be reconciled:

1. `(reconcile-declarative! component-ref)`
2. `(reconcile-direct! component-ref)`

The following will describe the mechanism, by which the lenses are able to reconcile the definitions of components to reflect changes applied through direct manipulation.

4.5.1. Compile Time Analysis

In order to preserve all the symbolic expressions within a component's render method, Transmorphic performs static analysis of the symbolic description at macro expansion time. For this, we make use of deep code walking macros [29] that traverse the entire nested symbolic expressions of the render method's lexical scope while performing various adaptations. This static analysis phase traverses the symbolic representation of the component's render method in two phases: An initial breadth first downwards traversal where analysis is performed, followed by an upwards traversal which instruments the symbolic descriptions based on the results of the previous analysis.

During the initial downwards traversal, Transmorphic handles the *tagging* of functions, assignment of *source locations* and the extraction of *source templates*.

Tagging A function is tagged during static analysis if it has been identified to yield either a morph or a component. This is determined based on resolving the names of the called symbols in the current scope and testing wether or not they map to a morph or a component function. Transmorphic is able to perform these tests, because every instance of `defmorph` or `defcomponent` causes the environment to register the respective abstraction together with its namespace as a morph or component respectively.

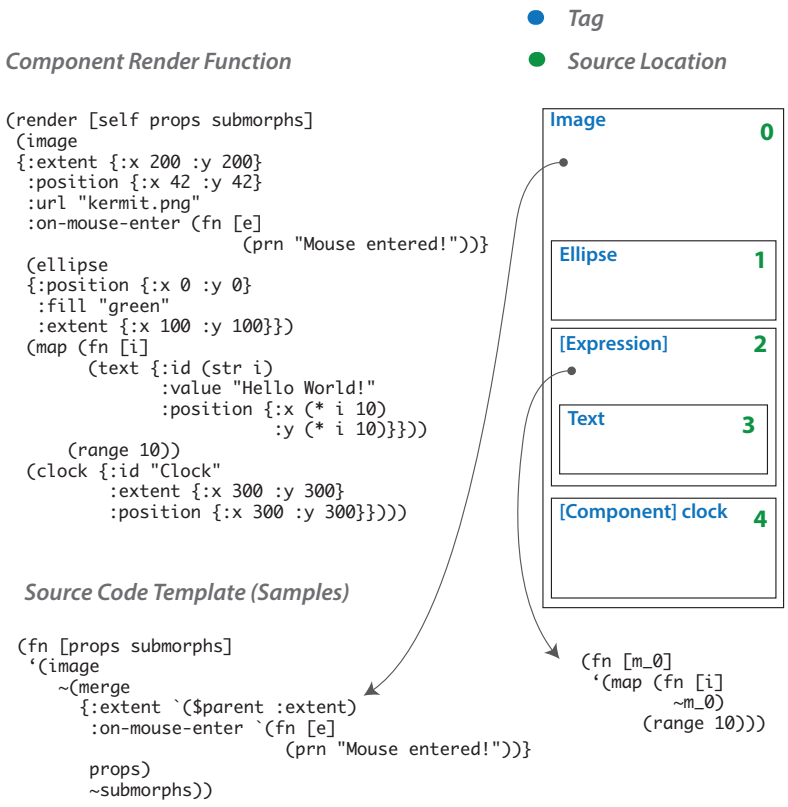

Figure 4.5.: Visualization of the static code analysis that happens at macro expansion time. The different stages, *tagging*, *source location assignment* and *source template extraction* are exemplarily shown by means of the *information artifacts* they produce.

Source Locations In addition, each tagged morph or component is assigned a static source location, which is later also attached to the rendered morph or component at runtime. Source locations are derived from a very simple enumeration scheme, based on a counter starting at 0 (for the root) that is then incremented each time a function is tagged. The source location allows Transmorphic to determine which part of the symbolic description is responsible for a certain morph or component, and alter the source at that particular location accordingly during the reconciliation phase.

Source Code Templates Lastly, the static analysis also includes the extraction of code templates, which are used to preserve symbolic expressions such as map, loop or anonymous functions. Code templates are extracted when during traversal the static analysis encounters a structure that is neither a morph nor a component. In this case Transmorphic retrieves a copy of the symbolic expression, replaces all of the calls to morphs or components inside with placeholder variables, and from there derives a function that can replace these placeholders with the reconciled calls to morphs or components. Before the downwards traversal continues, a map is assembled that associates the current source-location of the symbolic expression with that template function. Internally, Transmorphic refers to this stored mapping as the *reconciler*, which once finished, is associated with the component to ensure that it is store inside the key value store at runtime. Reconcilers are created and refreshed every time the compilation cycle and therefore the macro expansion phase is initiated again. Once this part of the reconciler is constructed, the analysis continues with all of the morphs or components that appear within the symbolic expression we just analyzed (basically the ones we replaced with the template variables in the code templates). Here we will apply the same procedure checking for morph, component or symbolic expression and adapt the reconciler accordingly.

Once the downwards traversal is finished, we end up with a reconciler that captures all of the symbolic expressions within the scope of the component's render function, together with a list of source locations that we assigned to the morphs and components. In the now initiated upwards traversal of the symbolic description, we transform each functional call within the component's render function that creates an entity, in such a way that it knows about its source location. At the same time the reconciler that has been extracted from the analysis phase is inserted into the component definition, such that in case the component is getting rendered, it has access to the reconciler. An example for a complete reconciler that is derived from a component definition, please take a look at Listing A.6 in the appendix.

4.5.2. Reconciling Morphs and Components

Based of the information that is extracted at compile time, we are able to access each morph's or component's source location and also the reconciler that is associated with each component. We will now explain the steps that the `PUTBACK` operation undertakes, in order to convert the original symbolic description together with the scene graph into an updated symbolic description.

In order to understand the reconciliation process, we first need to differentiate between two types of reconciliation, namely *external* and *internal* reconciliation. By means of an external reconciliation, we map a change back to the symbolic description based on changing the parametrization of the function call of the affected morph or component. This is essentially the set of transformations that were demonstrated in the previous chapter, where we described the `PUTBACK` behavior of the different lenses, i.e. removing, adding or adapting function calls to morphs or components. External reconciliation by itself however, is not sufficient for the transformation to take effect, since its independent of the lexical scope, namely the owner's render function that it resides in. We therefore also need a reconciliation step that reconciles the owner's definition, which is what the internal reconciliation achieves.

External Reconciliation During external reconciliation we translate the transactions applied to the respective morph or component into transformations inside the function calls. Note, that this does not yet affect the actual implementation of a component or morph, hence the term *external*. This is the stage, where the different reconciliation strategies come into play: In case we reconcile with the *direct manipulation* approach we check wether a source location is shared by more than one entity. If this is the, case we need to isolate the change with the conditional to only the entity where the transaction actually applied to, see Figure 3.8. If we follow the *declarative* reconciliation, we do not check for shared source locations and instead let the transaction that was applied most recently effectively override all of the previous transactions, see Figure 3.6. The external reconciliation of an entity always stops at the respective submorph collection, where we will determine which of the submorphs got added or removed, yet not bother with the actual reconciliation of these entities. Also note, that we only need to apply external reconciliation, to those entities that have actually been directly manipulated.

Internal Reconciliation After each component or morph has been reconciled externally, we continue with the internal reconciliation, which is the redefinition of the component's render function. This is the phase, where the previously extracted reconciler finally comes to use. Given a component that owns at least one entity which got transformed in the external reconciliation step, we first fetch that component's reconciler. We now recursively pick a reconciliation function from the reconciler, beginning with the one at source location 0 and evaluate it together with the external reconciliations of all the entities that are comprised by the symbolic expression at the current source location. We then continue the recursion with the submorphs of the externally reconciled entities until all entities comprised by the render functions scope are incorporated into the reconciled definition. Internal reconciliation propagates upwards, till the root of the world is reached.

4.6. Bypassing Compilation

The way Transmorphic maps direct manipulation to transformations inside the symbolic description may be elegant yet if applied naively, often leads to severe performance degradations when interacting with the system. The frequent recompilation of symbolic descriptions together with the separation of compilation environments constitutes a significant bottleneck during execution that can lead to a very unresponsive graphical representation. Especially in cases, where consecutive updates happen quickly in a row, it is sensible to devise strategies that allow us to postpone recompilation to a later point in time, when performance is not as critical.

We therefore enhanced the render pass such that Transmorphic is able to postpone recompilation as long as all changes are induced via the direct manipulation API. The ways in which the render pass of Transmorphic is optimized, can be split into three different measures:

Immediacy All direct manipulations that belong to morphs will immediately influence what the virtual DOM will be rendering.

Interception All direct manipulations to components will intercept the parametrization of the component at future render passes, emulating the effect the recompiled version of the component would have had.

Preservation All direct manipulations belonging to either morphs or components, are being preserved over the render passes that update the scene graph.

Changes to a morph or component are discarded, once the corresponding part of the symbolic description is updated. An update may happen by changing and recompiling the initial source files of the project, or by reconciling the symbolic description with the direct manipulation followed by a recompilation.

The concept of storing subsequently applying imperative adaptions after each render cycle is very closely related to the idea of *shaders* [10] in the domain of computer graphics. Similar to direct manipulation changes, shaders operate on the vertices, or pixels that have previously been produced by the render pass, in order to apply adaptations which can not easily be expressed in a purely functional projection.

4.6.1. Manipulation Transactions

Transmorphic separates the different mutations into three different types of transactions: Removing Transactions, Adding Transactions and Property Transactions.

Properties Changes to properties are independent to the order in which they were applied and stored directly in the transaction collection of the respective holder of the original properties. A property change will always strictly override the original value of the property, also in cases, where the property was initially declared dynamic or varying over time.

Removes If an entity is removed, the removal transaction is actually stored in the parent and not in the entry of the removed entity. In case an entity is removed, that once was added through an addition transaction, we do not insert a removal transaction but instead just remove the transaction that caused the entity to be added in the first place. This convention allows us to apply removal transaction without concerning ourselves with the order in which they were initially applied. Removal transactions are stored as a set of UUIDs, specifying the entities that are to be removed from the submorph array of the entity holding the removal transactions. When a removal transaction is registered, the removed entity is also orphanized. As described in the previous chapter, this will detach the entity from its original lexical scope, and prevent that future re-render cycles will be updating the stored entity in the key value store.

Additions The only type of transaction where order is important, stored as a consecutive list of UUIDs to entities that are meant to be concatenated to the existing collection of submorphs.

4.6.2. Consolidating Morphs

When we want to postpone the costly recompilation of the symbolic description, we need to ensure that the stored direct manipulations are being preserved throughout future render passes. We call the process that preserves these, without touching the symbolic description, the *consolidation phase*. In Transmorphic, the consolidation is achieved by interpreting each enacted meta operation as a transaction that is constantly replayed, every time the morph scene graph is re-rendered. Conceptually, transactions are therefore not influencing the rendering process itself, but something that is attached as a decorator to the result of the rendering process. Transactions are registered once one of the mutation functions is called, which causes the respective transaction to be placed into the transaction collection of the affected morph or component in the key value store. Transmorphic then ensures that, in every re-render cycle, the transaction collection is maintained and re-applied to the morphs and components in order to receive a consolidated morph scene graph that can be finally rendered by React.js.

4.6.3. Consolidating Components

When consolidating components, we first apply the same steps as in the case of the morph consolidation described previously, which includes the preservation of transactions and the corresponding adaption of the submorphs and properties belonging to the component. However after the application of the transactions, the component is parametrized with a possibly entirely different set of properties and submorphs, which requires the render method to be evaluated again together with the new parametrization. The result that is retrieved from the newly evaluated render method is then stored within the key value store and the consolidation process continues on the now updated submorph hierarchy of the component. Since every

parametrization of the render method may yield a unique collection of submorphs, Transmorphic further ensures that none of the once yielded submorph hierarchies are discarded, since they may at some point serve as the ingredient for other transactions applied to other morphs or components in the scene graph. In the case of long running sessions, Transmorphic may in fact accumulate a rather large amount of entries in the key value store, making the introduction of garbage collection mechanism necessary in order to prevent the exhaustion of memory. However in the current prototypical implementation of Transmorphic, we have not yet addressed this problem, making it part of the future work with regards to this project.

4.6.4. Propagating Changes

Besides emulating the effects of a changing parametrization for a morph or component respectively, we further need to be able to emulate the behavior that is created by symbolic expressions such as map statements. Fortunately we can make use of the a morph or component's source location that was assigned in the analysis phase for the purpose of reconciliation. In the context of the reconciler, "created by the same function call" essentially means that a group of morphs or a group of components share the same source location respectively. This makes the reconciler an ideal place to also store the information for change propagation that may become necessary when we want to skip compilation in response to direct manipulation.

Reconcilers therefore also store the same types of transactions derived from the mutations as the different entries for morphs and components in the key value store already do. The difference is that, within the reconciler, a transaction collection is not stored *per entity* but instead *per source location*. Despite being schematically equivalent to the transactions that are associated with entities, the transactions associated with source locations have the property of amplifying the respective transaction among all entities that share the same source location. Whenever the reconciliation mode for an entity is activated, Transmorphic will choose to apply the transactions corresponding to the source location instead of the changes associated with the actual entity.

4.7. Interfacing with React.js

Once the consolidation and the reconciliation phases are finished, the final scene graph is generated and then converted into a composite of React components that can be passed to the React interface. The extraction of the scene graph, happens by starting to traverse the key value store at the specially designated root entity, and from there recursively resolving all UUID references inside the consolidated submorph arrays. Since React.js already performs all the book keeping to transition the browser DOM as efficiently as possible, we do not need to worry about confronting React with a completely regenerated version of the scene graph.

5. Using Transmorphic and Outlook

In the following we will show how Transmorphic supports the programmer in various ways when developing an application by comparing an example development workflow to equivalent ones in Lively Kernel and Squeak. In particular, we will demonstrate the class based approach in Squeak/Smalltalk and the object based approach (i.e. "Parts") of developing an application in Lively Kernel's Morphic, and contrast this to the way it is done in Transmorphic. Each approach will be presented by examining different scenarios that may arise when developing an application. This includes the initial creation of an application, the incorporation of existing libraries or abstraction, the reuse of the application in a different context and finally the reconciliation of different changes by different users to the application.

5.1. Building a Clock

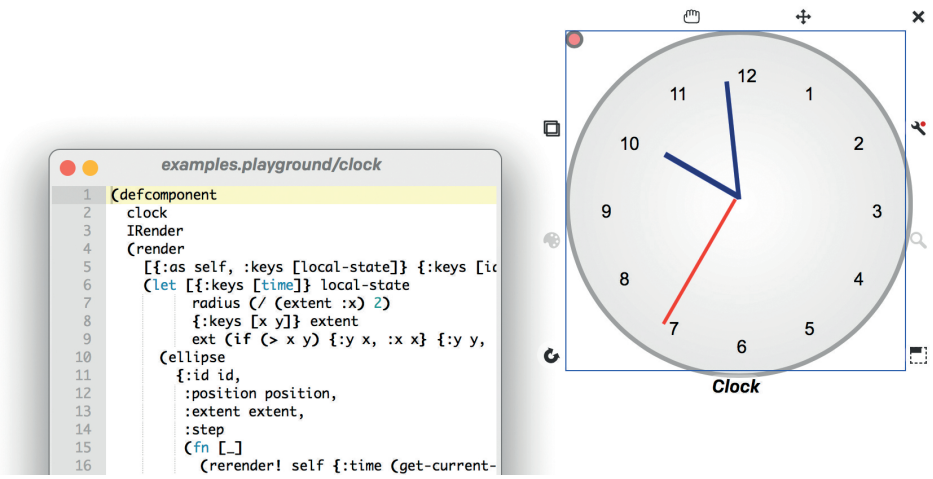

Figure 5.1.: Screenshot of the finished clock together with an open session of the Function Editor on the corresponding component definition

As an example application, we will create an analog clock that shows us the current time of the day together with the current date of the year. While this example may be simple, it still covers all the important aspects about a development scenario in Morphic within a nutshell:

1. **Assembly** Body and hands of the clock are suitable to be assembled by manual composition, while the placement of the labels for the different hours will be easier when generated programmatically. This will show how the different approaches are able to combine symbolic and direct manipulation approaches when defining the morph scene graph of the clock.

2. **Incorporating Abstractions** The widget that indicates the current date, is an already existing application that we will want to reuse in our clock and will allow us to compare how the different approaches allow the developer to incorporate existing abstractions.

3. **Behavior and State** We will then use the stepping mechanism of Morphic to evolve the clock's internal state, which also demonstrates how state can be managed.

4. **Collaboration** Lastly we will let a different developer enhance our clock by adding a checkbox that when selected, will let the hands of the clock go counterclockwise. The changes are then supposed to be reconciled with changes that we have applied to the clock in the meantime. This will demonstrate how behavior and appearance can be altered at runtime and then shared with other developers.

5.2. Building a Clock in Transmorphic

In the following we will illustrate, how the reconciliation process of Transmorphic modifies the source code of the clock component we are about to define. We will highlight the automatically changed parts of the source code in *orange*. Changes due to editing of the source code, meaning by the programmer, will be highlighted in *blue*.

During the process of developing the clock component, we will make use of a couple small tools that were not mentioned when presenting Transmorphic earlier. This includes the *Namespace Viewer*, which allows us to visually browse a certain namespace by rendering samples of each defined component in that Namespace. Each of the rendered samples can be grabbed and dropped into the scene in order to incorporate it into new component definitions. We will also make use of a very simplified Git [37] integration, which allows us to commit and push or pull changes with regards to a certain component. Notice that *all* of the tools we will use from within Transmorphic are themselves components defined in terms of morphs. We think that this also demonstrates the wide range of programs that are possible to create from within Transmorphic, indicating its universal applicability similar to Squeak/Smalltalk or Lively Kernel.

5.2.1. Assembly

We start building the clock by evolving an existing ellipse morph in the scene, using the halo to change appearance and shape to our preference and then opening a Function Editor in order to create a new component definition. After naming and saving the component definition, it will be compiled and hot swapped by Transmorphic. The upper right hand corner of the Function Editor will notify us, when this process has been completed.

Listing 5.1: The initial component definition of our clock. Notice that the properties such as `:extent` or `:position` are automatically merged into the properties of the ellipse which forms the root of the component. They therefore do not need to be specified in within the render function of the component.

```
1  (ns demos.clock)
2
3  ...
4
5  (defcomponent
6   clock
7   IRender
8   (render
9    [self props submorphs]
10    (ellipse
11     {:border-width 4,
12      :border-color "darkgrey",
13      :fill
14      "-webkit-gradient(radial, 50% 50%, 0, 50% 50%, 250,
15        from(rgb(255, 255, 255)), to(rgb(224, 224, 224)))",
16      :pivot-point {:x 0, :y 0}})))
```

We then continue with creating additional shapes required for our clock, such as hands and the labels which will represent the hours of the day. The hands will be constructed out of polygon morphs that we will define through a convenient halo interface that allows us to position the coordinates of the polygons vertices accordingly. This poses an example of the halo providing specialized functionality, depending on the type of the morph we are inspecting.

The label we will represent by a textmorph which we will stylize and place onto the surface of the clock as well. Thanks to continuous reconciliation, the symbolic description of our component has been updated accordingly such that we can now save and compile the component to start working on the loop, which will render the labels correctly.

The first thing we discover is that we might want to remove the complete definition of the hour label, to save some space and keep things more organized. We

Listing 5.2: Reconciled definition of the clock component, after hands and the sample of the label have been introduced

```
(defcomponent
 clock
 IRender
 (render
  [self props submorphs]
   (ellipse
    {:border-width 4,
     :border-color "darkgrey",
     :fill
     "-webkit-gradient(radial, 50% 50%, 0, 50% 50%, 250,
      from(rgb(255, 255, 255)), to(rgb(224, 224, 224)))",
     :pivot-point {:x 0, :y 0}}
     (text {:value "12"
            :position {:x -40 :y -5}
            :extent {:x 30 :y 30}
            :font-size 12})
      (polygon {:id "seconds"
                :fill "red"
                :vertices [{:x -10 :y 0} {:x 0 :y 0} {:x -5 :y 30}]})
      (polygon {:id "minutes"
                :fill "darkblue"
                :vertices [{:x -10 :y 0} {:x 0 :y 0} {:x -5 :y 30}]})
      (polygon {:id "hours"
                :fill "darkblue"
                :vertices [{:x -10 :y 0} {:x 0 :y 0} {:x -5 :y 30}]}))))
```

therefore select the label, open a separate Function Editor, and create an additional component which we will save and compile accordingly. Notice that Transmorphic handles all the necessary source changes for us, and the call to the morph functions is immediately replaced by a call to the component within our clock component definition Listing 5.3.

Listing 5.3: The reconciled source after the `hour-label` has been extracted and we manually wrapped it inside a map statement

```
(defcomponent
  clock
  IRender
  (render
    [self props submorphs]
     (ellipse
      {:border-width 4,
       :border-color "darkgrey",
       :fill
       "-webkit-gradient(radial, 50% 50%, 0, 50% 50%, 250,
        from(rgb(255, 255, 255)), to(rgb(224, 224, 224)))",
       :pivot-point {:x 0, :y 0}}
       (map
        (fn [hour]
          (hour-label
           { :id (str hour "h"),
            :label hour,
            :hour hour,
            :radius 150,
            :extent {:x 30 :y 30}
            :font-size 12}) )
        (range 1 13))
        (polygon {:id "seconds"
                  :fill "red"
                  :vertices [{:x -10 :y 0} {:x 0 :y 0} {:x -5 :y 30}]})
        (polygon {:id "minutes"
                  :fill "darkblue"
                  :vertices [{:x -10 :y 0} {:x 0 :y 0} {:x -5 :y 30}]})
        (polygon {:id "hours"
                  :fill "darkblue"
                  :vertices [{:x -10 :y 0} {:x 0 :y 0} {:x -5 :y 30}]})))
                     ↪ )
```

We can then wrap the call to the label component in a map statement that repeatedly renders the labels for each hour of the day. We will further pass the information such as position and the label contents as parameters to the component. Note that the position is immediately interpreted correctly, since properties are merged by

default. In order for the label property to start having an effect, we need to switch to the label component definition, which is easily reachable via halo and define that the label property influences the contents of the textmorph Listing 5.4.

Listing 5.4: The definition of the extracted `hour-label` component. Custom adaptations through editing are highlighted in blue.

```
(defcomponent hour-label
  IRender
  (render [self props _]
        (text
         {:id (str (props :label) "h")
          :position (point-from-polar
                     (* (props :radius) .8)
                     (angle-for-hour (props :hour)))
          :text-string (props :label)
          :font-family "Arial"})))
```

Lastly we want to fine tune the parametrization of the loop, which can be easily done by scrubbing the value inside the Function Editor. Transmorphic immediately compiles the map statement with the newly parametrized value, giving us a more direct feedback while we determine which value is best for the radius parameter. Notice that when we decide to adjust visual properties of the label in hindsight, we can do so without the loop abstraction being removed, by activating the declarative reconciliation and resizing one of the yielded labels to our liking.

5.2.2. Incorporation of Abstractions

As well behaved programmers we always strive to build on top of work that is already done, so incorporating external, already existing components is a very common use case. In Transmorphic, we are able to incorporate existing components (such as the date indicator) into our application, through either the direct manipulation or textual editing. In case we know the exact name of the component function, we can insert a plain call to the component into our symbolic description, which once saved, will cause the date indicator to appear within the clock. We can now go ahead an adjust its visual properties, which are automatically reconciled with the symbolic representation we have just updated. The other option is to perform incorporation via direct manipulation, and just grab and drop an already rendered sample of the date indicator into our clock: From the Namespace View that is opened on our namespace `demos.clocks`, we can grab the date indicator, and drop it into our rendered sample of the clock. With reconciliation enabled in the function editor, the required function call with the needed parameters will be placed in the source code automatically Listing 5.5.

Listing 5.5: The call required for the data-view, placed and parametrized accordingly in the symbolic description of the system (highlighted in orange)

```
(defcomponent hour-label
  IRender
  (render [self props _]
        (ellipse { ... }

          (date-viewer
           {:position {:x 30 :y 40}})))
```

5.2.3. Behavior and State

In Transmorphic the behavior and management of state are cleanly separate from the visual representation. This means that no part of the logic concerned with data management will ever reference a visual component directly, while the visual elements themselves should not be strongly coupled to the application state. We will therefore not mix code that causes the hands to change with code that updates the current time of the clock. The root of our component will be supply with a `:step` callback which will compute the current rotation values for the respective hands as seen in Listing 5.6.

Listing 5.6: The step method we will attach to the root of the component. In principle, any entity that is owned by the clock component could implement the `:step` method, yet it is most convenient to pick the root of a component to serve this purpose.

```
{ ...
  :step (fn [_]
        (swap! self assoc :time (get-current-time))}
```

We then wire up the rotation properties of the hand with the values inside the state, so that the rotation of the hands always represents the value of the current time, as seen in Listing 5.7. Currently, all of this needs to be done through manually editing the source code and triggering a recompile of the component definition.

When it comes to sharing our application with other Transmorphic environments, it suffices to share the symbolic description that is required for clock we have just created since all our changes are reflected in the symbolic domain. Thus, the sharing of our clock can work with any type of text based versioning system. In fact, Transmorphic provides a simple interface to interact with a local git repository, which is accessible via the halo. From here we can commit our changes and share our new clock component with other users.

Listing 5.7: Excerpt of the clock component definition, where the hands `:rotation` property has been hooked up to the component local state (highlighted in blue)

```
...
(polygon {:id "seconds"
          :fill "red"
          :rotation (* (+ -0.25 (/ (self :seconds) 60)) 2 PI)
          :vertices [{:x -10 :y 0} {:x 0 :y 0} {:x -5 :y 30}]})
(polygon {:id "minutes"
          :fill "darkblue"
          :rotation (* (+ -0.25 (/ (self :minutes) 60)) 2 PI)
          :vertices [{:x -10 :y 0} {:x 0 :y 0} {:x -5 :y 30}]})
(polygon {:id "hours"
          :fill "darkblue"
          :rotation (* (+ -0.25 (/ (self :hours) 12)) PI 2)
          :vertices [{:x -10 :y 0} {:x 0 :y 0} {:x -5 :y 30}]})))
```

5.2.4. Collaboration

In the following we will describe how two different users, namely *Alice*, operating on the system we have worked on so far, and *Bob* who will be working in a separate instance of Transmorphic that has not yet retrieved the clock component. Bob can get access to the clock's implementation by retrieving the changes committed previously by Alice, via some text based versioning system into his local repository. After this is completed, we can, again, make use of the Namespace View in order to grab the clock we just received, and start to evolve it by selecting the component with the halo and starting up the function editor, see Figure 5.2.

We first want to change the shape of the root of the clock, so we simply alter the symbolic description a little bit, and replace `ellipse` with `polygon` instead. After compilation the shape of the clock switches to a triangle, which is the default polygon shape when we do not supply any vertices. We can now shift back into direct manipulation by using the halo interface for vertex manipulation of the polygon, and add more vertices to the polygon, altering its overall shape, see Figure 5.3. Again, by triggering the reconciliation inside the Function Editor, we are able to incorporate the source code transformations due to the added or changed vertices, see Figure 5.3.

Meanwhile, Alice decides to further extend the behavior of the clock as follows: She first adds the checkbox onto the clock by grabbing and placing it at the appropriate position. Shifting back into the symbolic description of the clock, she implements the callback inside the :on-change behavioral property of the checkbox, which will in its turn modify the state of the clock to now turn in the opposite direction, see Listing 5.8.

She will also add a text morph that labels the checkbox to convey the effect it will have on the clock's behavior and alter the order in which the checkmark box is rendered, since she does not want it to cover the hands of the clock.

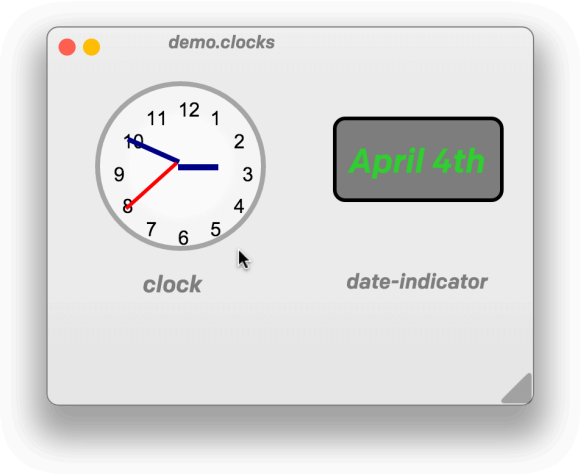

Figure 5.2.: Within Bob's Transmorphic instance, we are able to retrieve the new clock component via the Namespace View, once the changes have been pulled.

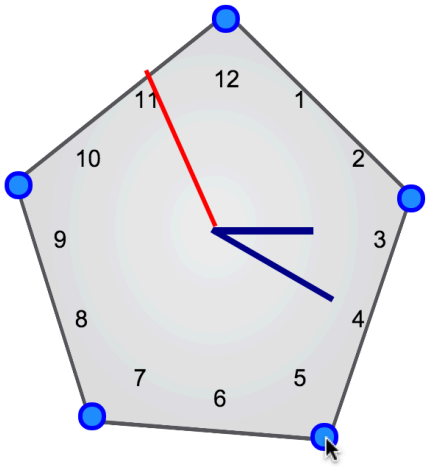

Figure 5.3.: Adapting the polygon vertices of the new clock root, through direct manipulation

Listing 5.8: Manual changes by Alice, needed to implement the behavior of the checkbox

```
(checkbox
 { ...
  :on-change (fn [checked?]
               (swap! self assoc :backwards? checked?)
   ... })
    (let [direction (if (self :backwards?) 1 -1)]
      (polygon {:id "seconds"
              :fill "red"
              :rotation (* direction (+ -0.25 (/ (self :seconds) 60))
                 ↪ 2 PI)
              :vertices [{:x -10 :y 0} {:x 0 :y 0} {:x -5 :y 30}]})
      (polygon {:id "minutes"
              :fill "darkblue"
              :rotation (* direction (+ -0.25 (/ (self :minutes) 60))
                 ↪ 2 PI)
              :vertices [{:x -10 :y 0} {:x 0 :y 0} {:x -5 :y 30}]})
      (polygon {:id "hours"
              :fill "darkblue"
              :rotation (* direction (+ -0.25 (/ (self :hours) 12)) PI
                 ↪  2)
              :vertices [{:x -10 :y 0} {:x 0 :y 0} {:x -5 :y 30}]}) )
```

Notice how up till now Alice has switched multiple times between both visual and symbolic domain, almost effortlessly, regardless of changes to behavior, state or appearance of the morph. The only requirement was that in between the symbolic description got saved and recompiled, which however does require only little effort from the perspective of the programmer. Alice and Bob finally commit their updated clocks, via the halo menu in order to make it accessible for other users as seen in Figure 5.4.

Since now the branch that Alice and Bob have both worked on has diverged, they will want to combine their changes. Fortunately all changes are reflected in the clock's component definition, and although Bob completely replace the body of the clock, and Alice altered its behavior, both changes are completely independent from each other: For example, if Alice decides to reject Bob's visual enhancements it is "easy" for her to differentiate, and to exclude the lines form the merge while still preserving the checkbox including the altered behavior, since these are reified in a completely separate part of the source code.

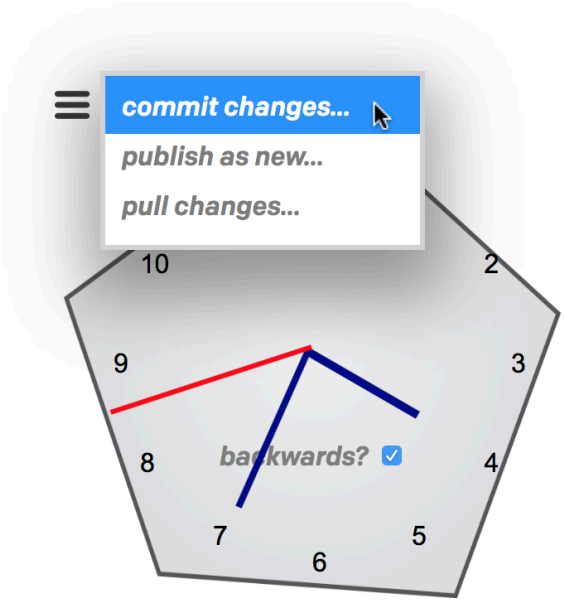

Figure 5.4.: Committing the changes applied to the clock via the simplified GIT integration accessible through the halo

5.3. Comparison to other Morphic Implementations

We will now contrast the previously described construction of a clock in Transmorphic with two alternative ones possible in Lively Kernel and Squeak/Smalltalk respectively. We will not repeat the required steps of each workflow section in detail, and instead just emphasize the differences to the approach presented in Transmorphic.

5.3.1. Class Based Approach (Squeak/Smalltalk Morphic)

Morphic in Squeak/Smalltalk [19] is based on classes and objects, which means we develop the clock by working from the symbolic domain instead of starting in the visual domain. This requires the programmer to perform the back link from visual representation to code manually. We will frequently recreate an instance of the clock by hand in order to check if our changes to the code have taken the desired effects. This helps in reducing the possibility of transient errors within the state of the application, since at creation the state of the clock is always "untouched". The class based approach also provides various benefits, when different users want to collaborate different changes over time, since all changes are entirely symbolic and therefore text based. We will not present each stage in detail as we did in the case of

Transmorphic, however the whole definition of the clock morph can be found in the Appendix at A.4, A.5.

1. **Assembly** When defining the clock based on an entirely class based approach, we will most likely start out by defining a class the serves as a base for our desired clock morph, such as a subclass of `EllipseMorph`. We will then continue defining the visual appearance of the clock, by evolving the `initialize` method of our clock with various subroutines that handle the placement of the labels and the hands. Assembling a clock in this approach requires the us to manually retrieve visual feedback whenever we want to check how the symbolic description actually affects the end result. This will lead us to either switch constantly between experimenting with manipulations of the interface through the halo, reading off the corresponding values, and adapting the symbolic description accordingly, or just defining everything in terms of symbolic relationships, visualizing the eventual result in our head. Compared to Transmorphic, we will need to spend more time on this stage of the workflow, since there is no mechanism that allows us to quickly extract symbolic descriptions from the graphical representation.

2. **Incorporating Abstractions** When we want to reuse an existing abstraction that already provides an interface we need (such as the date indicator), the class based approach requires us to manually track down the symbolic name for that abstraction, and also its required parametrization. Fortunately Squeak/Smalltalkprovides various means to quickly determine the types of visual objects or search the source for certain terms that may lead us to the corresponding abstraction. Once we have found the class corresponding to our date indicator, we just enhance the initialization routine with a call to the date indicator's constructor. By specifying the dependency solely in the symbolic description of the clock, we guarantee that we always spawn the clock with the most recent version of the date indicator as well, provided that our source base is up to date. Again, fine tuning of the visual parameters becomes manual work we have to perform by adapting the source and reinitializing the morph accordingly.

3. **Behavior and State** Management of behavior and state does not differ significantly compared to Transmorphic, since this constitutes the part of the clock where we do not get to benefit from direct manipulation at all. The implementation strategy is essentially the same, namely implementing the `step` method wherein we ensure that the hands are updated accordingly each time the clock is being stepped. A slight difference is that the hands, unlike in Transmorphic, will be altered via side effects; that is, explicitly setting their rotation property to a certain value inside the update routine. This is a less declarative and more imperative style to define updates in the visual representation of the clock and does not present the relationship between state and appearance as clearly.

4. **Collaboration** Similar to Transmorphic, long term collaboration is simplified since everything about the clock is expressed in terms of symbolic descriptions

and we do not have to concern ourselves with versioning state. For example, changing the shape of the clock's body, can be done by changing the superclass of the clock to a `PolygonMorph`, while the behavior can be altered through the introduction of new methods, and the incorporation of a button morph, which allows us to toggle the moving direction of the hand. Reconciling the changes from Bob and Alice respectively, boils down to the same situation described in Transmorphic, where we can precisely pick which changes we want to incorporate and which to reject.

5.3.2. Parts Based Approach (Lively Kernel)

The parts based approach completely embraces direct manipulation authoring, meaning that the programmer specifies the visual representation entirely through assembling and mutating objects, i.e. *instances* of morphs. When working with parts, we focus on the visual decomposition of a graphical user interface, where modularization is defined by the visual entities the interface consists of. Being plain objects in memory, parts can not be shared directly, but need special tool support that either serializes them into another representation (i.e BuildSpecs) or provides an interface to perform change management on the parts that are shared between different environments.

1. **Assembly** In order to assemble the clock through parts, we mostly rely on directly manipulating and copying of visual elements. We will only consider working in the symbolic domain, when we determine that the manipulation by hand, would turn out to be too tedious, as is the case with the placement and alignment of the hour labels. In fact we can use Lively Kernel's workspace, to evaluate arbitrary code that helps us to change the appearance of our clock in certain ways. A change once applied to one of the labels however, is not automatically propagated among all the others, since each label is a separate, independent morph. This requires us to manually clear, recreate and layout the labels, each time we perform an adaption to one of the labels. In cases where we frequently use custom setup code for certain elements, we should start thinking about including this setup code into the behavior of the application, in order to let other users easily restore the original layout of the elements, or support them when they want to adapt the custom layout to their preference. We will also want to assign names to each part that we may want to reference in the future, since this allows us to easily retrieve references of these parts later on.

2. **Incorporating external Abstractions** Similar to the parts we have been using to construct the current element of the clock, also the data indicator exists in the form of a part within Lively Kernel's *Parts Bin*. We can just drag it from there into the clock, and position it accordingly. Lively Kernel is able to internally track the history of parts, and will allow the user to update parts in case new versions of them have been released by the author. Therefore an update to

the date indicator can be merged into the clock, by using the morph menu reachable via the halo.

3. **Behavior and State** When working with parts in Lively, we can easily enhance their behavior through the object editor, by selecting the morph that we want to evolve via the halo and opening the Object Editor through the edit menu. Here we can directly implement the step method similar to the way we did in the class based approach. When updating the hands, we can refer to them by retrieving a reference via their respective id, or by storing that reference inside a member variable that belongs to the clock part itself and then using the member variable instead.

4. **Collaboration** After the clock has been completed, we can share it as a new part, which means that we essentially share the state of the clock with other instances of Lively Kernel. Other users can now retrieve the clock via the Parts Bin and use it in their applications or enhance the clock, as we will do now. Again we will employ the same halo interactions, as we did in the case of the class based approach, while not concerning ourselves with a symbolic description of the appearance.

The loss of object identity in the symbolic description in the case of Transmorphic, becomes a benefit in the context of long term collaboration, since conflicting or diverged changes on the same entity are in the end always resolvable by reconciling text. Reconciling changes from different users to a part, can therefore become difficult, since we need collaborate on the serialized representation of parts and not a purely text based, symbolic description. For example, when Bob changes the appearance of the clock's body, he will most likely replace the complete clock, copying the behavior while also transferring labels and hands to the new body. From the perspective of object identity, it now becomes more challenging to tell what stayed the *same* and what actually *changed*, since essentially the whole clock was replaced. From a conceptual point though, what only changed was the shape of the clock, while everything else stayed the same, however this needs to be inferred manually or with the support from tools that allow to compare and merge serialized parts with each other. Lively Kernel provides an alternative, document like description for morphs, called the *BuildSpec*, which can be automatically retrieved for each morph. BuildSpecs remove the object notion of parts, and transform them to a purely declarative description very similar to the way Transmorphic specifies morphs. Through this mechanism, it again becomes easy, to differentiate between Alice and Bob's respective changes, and combine them accordingly. Unlike Transmorphic, BuildSpecs do not preserve abstractions responsible for the visual representation, which makes them more space consuming than the component definitions used in Transmorphic and also does not allow them to leverage relationships between different morphs that may arise from the symbolic description that spawned the morphs in the first place.

5.4. Limitations of Transmorphic

In its current form, Transmorphic also compromises on several aspects in the development workflow, some of which are inherent to the approach, and others which could be resolved in future implementations.

The first area, where Transmorphic is currently limited is the reconciliation of properties that are expressed by a symbolic binding: In cases where Transmorphic finds that a certain property has been changed, the symbolic binding within the scope will be directly replaced, without taking the actual derivation of that property into account. For example, reconciliation will not consider re-declaring local variables or slightly altering computations that led to a certain value, but instead just replace the variable that a property is bound to with a constant.

While this convention ensures correctness, it is in many cases not the most elegant solution. By always replacing the value of a property inside the call of a morph function, we require all properties to always be declared inline of the function call, and do not support the style of binding properties at one point to a variable, and then distributing them to different morphs or components through that variable. Transmorphic currently also does not preserve the custom formatting of the code provided by the user, but instead selects the pretty printed version of the reconciled symbolic description. Thus, symbolic descriptions that are being rewritten by the reconciliation mechanism are starting to look more "generated", overriding the author's custom code formatting.

We further found that while the way of extracting identities from the scene graph works in most cases, there are caveats to the current approach: While direct manipulation does not interfere with the identity concept (in fact it *requires* the identity to work), manual editing and recompiling components can lead to a completely different identity resolution, possibly invalidating all of the previously inferred UUIDs. As mentioned in previous sections, this requires us to remove direct manipulation changes from the scene graph, since we have not yet found a reliable mechanism which allows us to re-infer the identities of the entities.

Transmorphic's Function Editor currently does not support a way to simultaneously change code through keyboard input and incorporating reconciled symbolic descriptions. This problem could be circumvented, by enforcing a certain structure onto the code throughout the edit session, for example by turning the function editor text input into a *structural editor* [34]. However for now we refrain from turing the function editor into a structural one, since this would restrict the freedom of the programmer to change the symbolic description of a namespace or component.

On a more fundamental level, there are also restrictions that come from the fact that Transmorphic separates view and model very strictly from each other. This becomes clear, when we look at example programs in "classical" Morphic that model physical entities by directly using visual properties in order to influence the view as well as the model at the same time. One example for this would be a morph which models a particle that can be dragged and dropped by the user into various contexts. Once dropped inside a certain morph, the particle then automatically detects its owner's bounds and then continues to move freely while always bouncing off the owner's

bounds. Real world scenarios, such as this particle simulation, are very elegant to implement inside a Morphic where model and visual properties are combined in the same object. In Transmorphic however, the programmer is always required to provide a model that contains all the needed information to derive a view. The example of the particle morph could not achieve the same amount of universality in Transmorphic, since that would mean that all of the existing applications would need to be made explicitly aware to react to the ball morph once it is dropped on them. While Transmorphic preserves the close relationship between visual properties and state management, both are still strictly decoupled which reduces complexity but also flexibility of the view's implementation.

5.5. Future Work

5.5.1. Time Travel

Since all information in Transmorphic is stored in persistent data structures, we can keep a continuous history of how component local state, symbolic description or the morph scene graph evolved over time. A mechanism that allows us to revert each of these data structures respectively is therefore easy to implement, however one also needs to provide a meaningful interface for the programmer, to incorporate this time travel mechanism into the actual workflow. Also the question arises, wether these three internal data structures in Transmorphic should be reverted independently from each other, of if the history of user interactions should always be kept in chronological order. While independently reverting state and symbolic description may provide powerful means for the programmer, to experiment with behavior or save time and compare different scenarios, it can also lead to confusing situations which would be prevented by enforcing the chronological order of events.

5.5.2. Improving Reconciliation

Currently Transmorphic provides the *direct manipulation* and the more *declarative* reconciliation approach. There are however several ways in which the existing reconciliation mechanisms of Transmorphic can be further improved: For example what may be solved by the conditional in case of a simple loop statement, may be solved differently in the context of a let statement, where in case of removal, we may want to just remove the binding and all the statements the bound variable ever occurred in. Going even further, there are different conventions we can imagine for actually placing the conditional, such as introducing them directly at the definition of the property, or instead wrapping the whole morph call inside a conditional and so on. Further diversifying the strategies with regards to the current symbolic expression, increases the quality of reconciled code, and eventually provides the programmer with reconciled symbolic descriptions that can be included into the source code with little to no corrections necessary.

Furthermore, the reconciliation interfaces, such as the Function Editor, could also be improved further: Leaving the programmer complete freedom when editing the symbolic description comes at the price that throughout manual editing of the symbolic description, we allow code to be syntactically, semantically incorrect. As mentioned, this forces us to limit the incorporation of reconciled code to a separate mode where user input is turned off. A partial solution to circumvent this problem could be, to leverage the nested list structure of lisp expressions: For example, we could imagine a function editor that keeps track of the existing nested lisp expressions, and continues reconciliation throughout the code being edited, yet only for those parts of the symbolic description that have not yet been affected by edit operations. This would decrease the need to actively switch modes every time the programmer wants to incorporate the direct manipulations into the source code.

5.5.3. Expanding Reconciliation

It is worthwhile to consider reconciliation mechanisms between symbolic and graphical representation that go beyond the domain of graphical user interfaces. For instance the derivation of a domain model from data, can also be thought of as an interactive process, where the programmer can define the decomposition of the data and domain specific logic, by interacting with the data through direct manipulation. Conceptually, we would translate the halo mechanism from a purely visual domain, to the domain of data decomposition, providing various tools to actually define the projection of data into visual entities, which currently still needs to be done completely by hand in the symbolic domain. The halo could further provide ways to manipulate visual representations of data flow inside the application, or data outside of the component local state, such as data residing on remote servers. Information could also be fetched in a much more declarative manner, by specifying queries that are colocated together with the respective components. The halo could then be used to dynamically specify queries associated with a component by direct manipulation, while also providing a visual feedback that displays the data that is retrieved by the queries.

We have seen declarative constructs such as relative properties, and how Transmorphic is able to propagate changes through the symbolic description of a morph collection. This gives rise to the question, of wether there are further declarative constructs that could be introduced to Transmorphic and thereby support the programmer. Examples for this include properties that are based on constraints, custom defined events that directly react to changes in the properties of a certain morph or abstractions that influence the layout of a whole group of morphs.

5.5.4. Realtime Collaboration

In the context of this work, we have only considered long term, asynchronous collaboration between users and how conflicts arising in symbolic descriptions can be reconciled. It is however also worthwhile to explore the potential of Transmorphic's declarative descriptions in the context of real time collaboration: The bidirectional

relationship between graphical representation and symbolic description, can also be used to synchronize different Transmorphic development environments in realtime. Realtime synchronization however also introduces similar questions that arose in the case of time travel, namely *what* information to actually synchronize between clients. Synchronizing the application state, scene graph and symbolic description, involves further investigation on how to isolate collaborating environments, since we run into conflicts with regards to code that assumes to be execute in a certain physical context. Here we could include solutions provided by the Croquet project [32] which encountered the same problems, while implementing a decentralized synchronization mechanism based on state replication.

We could however also just restrict synchronization to the actually rendered scene graph and component local state: This would have the benefit of removing the problems that arise from application code that is execution context dependent, yet provide all of the look and feel that is required for meaningful collaboration in real time.

6. Related Work

6.1. Lively Kernel

The Lively Kernel is a live, self sustaining, Javascript based development environment, running in the browser. It combines a wide range of tools and frameworks for developing and sharing various kinds of web applications inside a single environment. Among other things, this includes an easily extensible client server infrastructure, seamless communication between different Lively environments (Lively2Lively) and an implementation of Morphic where the DOM can be directly manipulated and transformed through a Morphic programming interface.

Unlike Transmorphic, Lively embraces the imperative programming paradigm in Morphic which is derived from environments like Squeak [19] or Self [5]. In the context of this work Lively's object centric concept of Parts with its direct manipulation authoring is of particular interest. The nature of parts resembles the real physical world as closely as possible, enabling workflows that are very much like the crafting of physical objects in real life. The decision to use objects instead of a symbolic abstractions as the medium of collaboration has various implications: For one, the changes that a programer enacts are automatically isolated, since every change is contained by the current instance and not based on a symbolic abstraction that may be instantiated in various different contexts. On the other hand, object identity is a rather complicated basis for long term collaboration, since it requires that identity of objects is constantly resolved, when different parts are updated or combined. Efficiently collaborating with parts therefore requires a variety of tool support which includes basic sharing interfaces such as the Parts Bin [23] that also keeps track of the derivation history, preserves the modularity of the different parts, and allows the developer to update the constituent parts of an assembled application in case updated versions are being released. More advanced mechanisms for collaborations such as Lively Groups [12] allow to share behavior among different parts. There are also various approaches for realtime collaboration, providing mechanisms for comparing different versions of parts, and even synchronizing the whole state of remotely collaborating instances of Lively [4].

6.2. Easy Morphic GUI Framework [EMG]

The Etoys [20] direct manipulation based authoring scheme is similar to the object centric approach in Lively Kernel [23] yet it does not revolve around objects but instead uses Singleton classes to represent each entity. Similar to Lively, tool sup-

port is an essential aspect about Etoys, since the singleton classes of Etoys do not interface well with the usual source management tools such as the Class Browser. In Etoys, tooling will usually provide a structural editing interface in order to be able to map changes in the visual representation back into the respective singleton class definition.

The Easy Morphic GUI Framework (EMG), tries to combine the prototype based approach of developing applications in Morphic with the class based approach for implementing the Model logic of the application. Similar to Transmorphic, EMG mediates between two different domains, namely the class based, symbolic domain and the object based prototypes for the specification of the GUI. In EMG the class based symbolic domain is reserved for the *Business Logic* of the application, while the visual interfaces are still programmed using the singleton class approach from Etoys. EMG provides a default superclass for all of the Business Logic classes, which defines a default interface for talking with the Business Logic of an application defined in Etoys. The actual wiring of GUI to Business Logic can happen entirely through direct manipulation with EMG automatically generating the necessary code in the background,. This assumes that all of the business logic is implemented based on the EMG classes that come with EMG. Similar to parts in Lively, the sharing of the visual interfaces is solely based to serialization of the respective class objects. Compared to Transmorphic, EMG does not provide a complete symbolic description that encompasses visual as well as data specific aspects of the application.

6.3. Live Programming in Touch Develop

Microsoft's Touch Develop [36] programming language is tailored for developing simple ("non professional") applications on devices restricted to touch user input. *Burckhardt et al.* [3] provide an enhancement to Touch Develop that is able to establish a symmetric relationship between graphical and symbolic representation, similar to Transmorphic. Since Touch Develop is a structured and imperative programming language, establishing a reliable symmetric relationship between graphical and symbolic representation is challenging. Therefore, instead of imposing a functional description of the view like Transmorphic does, the mechanism presented in this extension of Touch Develop introduces restrictions to the procedures operating on view specific information:

1. Like in Transmorphic, the view is stateless, meaning that any entity that is visual can not be further modified once created. Changes can only be enacted by referencing values in the application state and re rendering the view, once user interactions have altered the state.

2. Render code can not mutate values at all and is only able to read information.

3. Code that is concerned with the application state, can not create visual elements. This restriction together with (2) enforces a unidirectional data flow from data to view in an otherwise entirely imperative language environment.

The programmer has to scope visual elements inside so called "box" statements, which allow the system to precisely determine, which part of the symbolic description is responsible for the visual representation of the application. In the end this approach achieves an effect similar to Transmorphic, in that we are able to translate changes in state into source code transformations. Note however that in the case of Touch Develop, the restrictions are not apparent from the symbolic description of the application, but need to be enforced by the runtime. In case side effects, or other "forbidden" operations happen to be executed in the context of code that provides the graphical representation of an application, evaluation is terminated.

6.4. QML

QML is a declarative description to describe graphical user interfaces in the QT Framework. The QML programming language is syntactically based on JSON, and allows to seamlessly integrate Javascript code into the document based description, which allows the programmer to combine imperative and declarative abstractions. Very similar to Transmorphic, QML presents each UI component by declaring the component type and also each of its properties respectively. In addition, QML provides various additional declarative constructs, such as the custom definition of events, the concept of relative properties and declaratively specification of animations.

Direct Manipulation authoring is also possible in QML through the QT Quick Designer, which provides a wide variety of tools to evolve the visual representation of a GUI while automatically keeping the symbolic description in sync. Editing in QT Quick Designer is automatically disabled, as soon as imperative abstractions are being used in the symbolic description of an interface and a reliable back mapping of changes in the visual representation to the symbolic domain becomes unfeasible. Compared to this, Transmorphic does not stop reconciliation in cases where side effects become present, though the reconciled code may turn out to be incorrect. In this regard QML is more restrictive but also more precise about what it can and can not do, which is necessary since QML is being widely used in various "production ready" commercial contexts. In addition to that, direct manipulation in QT Quick Designer or changes in the QML code in general are no applicable at runtime, but require the application to be recompiled and relaunched. There is a strict differentiation between compile time and runtime, which does not allow for an interface to be evolved at the actual runtime of the application.

6.5. Apparatus

Apparatus [30] is a recursive drawing tool, which allows the user to define visual and interactive diagrams. Derived from recursive referencing of prototypes in Sutherland's Sketchpad, the user is able to incorporate a definition inside a new definition

of an object. Apparatus evolves the idea of Sketchpad to an environment with a functional declarative description and a visual representation. Unlike Transmorphic, the declarative domain is not represented by code, but instead a separate visualization of the object properties, reminiscent of a structural editing interface. Since all of the objects are defined in a purely declarative way, unbound properties can be mapped back and forth, just as in Transmorphic.

In order to achieve interactiveness of the diagrams, Apparatus pursues a purely declarative approach, and does not resort to imperative abstraction like Transmorphic does. For instance, objects do not carry a set of functions defining their behavior (e.g. a step function) but instead define dependencies between the different properties, through variables. Variables define how the properties of the different objects in a scene relate to each other. Apparatus then starts treating the property to variable relationship as a system of constraints that is then solved with respect to the unknown variables. The user can specify what part of the relationship is supposed to be the unknown from the Apparatus interface, namely the free and fixed variables. In this way Apparatus allows the user to vary the way the declarative representation is interpreted by the system, thereby varying the interactive behavior of the objects. Compared to this, Transmorphic only enforces the purely functional nature of the morphs with regards to rendering the graphical representation, not regarding the application logic. Apparatus on the other hand requires the user to be aware of all datasource and also establish a functional dependency network between the sources and the values they affect. From a software architectural point of view, Apparatus does something desirable, namely it nudges the user into discovering a way to find a functional dependency between the inputs and the outputs. For an example of a Clock that relationship may be straight forward to find. However, given an interactive application, that interacts with the user in more complex patterns, the functional programmer needs to incorporate more thought into the underlying denotational semantics of the domain. In this case, mutable state is helpful, despite being more error prone, and the development of a prototype application is feasible in much less time. It is not possible to model procedural execution, conditionals or recursion in Apparatus, making it a non Turing complete programming environment. Therefore, in its current form, Apparatus is restricted to the domain of self contained diagrams that are based upon a mathematical model.

6.6. KScript

KScript is a mostly declarative, and dynamic language that uses the functional but time aware approach known from FRP to describe a large variety of an applications behavior. In KScript, dictionary like objects, equivalent to the ones found in Javascript, are combined with FRP style data flow programming. Unlike classical FRP, KScript does not enforce a purely functional description of the application, but rather encourages mixing imperative and functional abstractions in order to create brief and concise symbolic descriptions, that closely resemble the *intention* of the executed program.

The imperative programming model is also required in order to provide support for direct manipulation tools such as the halo. The mixing of imperative and FRP based abstractions greatly reduces the lines of code necessary to express a certain program, but gives rise to problematic runtime behavior: For example, imperative changes are not immediately propagated inside the graph of FRP values, since no observer mechanism is installed by default. This leads to partially inconsistent states inside the FRP signal graph, potentially leading to unexpected behavior. In practice, the mixing of FRP abstractions and imperative statements can lead to bugs that are very hard to track down, despite the reduced amount of code. FRP streams and behaviors are able to trigger code execution from multiple different starting points within a block of code, which can create a complexity problem that is similar to the one of carelessly used *goto* statements. While KScript does not provide a symmetric relationship between symbolic and graphical representation, it does present a possible approach to combining imperative and functional abstraction for GUI programming. However in order to be used in a meaningful way, further tool support needs to be devised that supports the programmer when dealing with bugs that may arise due to the combination of these two paradigms.

6.7. Elm

Elm [8] is a functional programming language for declaratively creating web browser-based graphical user interfaces. It employs a customized type of functional reactive programming [40] and similar to Transmorphic, performs its updates in the view completely based on functional projections. Elm is directly derived from Haskell, syntactically extremely close, and in many case exactly the same. Similar to Haskell, there is no use of mutations or other destructive behavior in Elm, which means that even the model domain of an Elm program is completely free of any side effect, effectively pushing the "imperative shell" of the program to signals that are part of the FRP model.

Elm tries to enable liveness through immediate and quick recompilation of the code, while also providing a very fast emersion since all of the state management is happening in a non destructive manner. The rigid functional frame, that comes with each Elm program however, currently prevents any form of direct manipulation authoring or inspection of visual entities. The programmer has to evolve an application within the symbolic domain at any time, though feedback is quickly accessible.

6.8. Sketch-N-Sketch

Sketch-N-Sketch [7] is a development environment for developing interactive vector graphics by combining direct manipulation and source code editing, similar to the way it is done in Transmorphic. Ravi Chugh et al. refer to this combination of domains as *Prodirect Manipulation*, since it allows direct (visual) as well as pro-

grammatic (symbolic) manipulation. Sketch-N-Sketch tries to preserve the symbolic description in a much stricter fashion than Transmorphic, making the direct manipulation in the graphical representation of the system quite different to the one known from Morphic. Sketch-N-Sketch comes with its own custom programming language called *little* which enables effective tracing of values inside the definitions of the vector graphics. This allows the system to preserve symbolic relationships in cases manipulation happens, while also allowing to vary how much of the symbolic description is supposed to be preserved. Similar to Apparatus, Sketch-N-Sketch is currently limited to stateless applications such as interactive diagrams. We could not find examples for interactive applications that provided services such as tooling for development environments etc.

7. Conclusion

We have presented Transmorphic which provides a functional and declarative way of describing Morphic applications. We identified the potential of combining imperative and functional concepts in the context of developing applications in Morphic. We took the mutable-abstraction-based Morphic as a starting point and turned it into a functional and declarative version, while trying not to compromise on the liveness and directness properties that are vital to the Morphic programming experience. To do that, we established a symmetric mapping between code and graphical representation, that is built upon the concept of functional lenses, which allows us to not only preserve liveness and directness in a functional environment but also provide entirely new perspectives on how a morph can be evolved at runtime.

Switching between the symbolic and the visual domain becomes instantaneous. Abstractions in one domain and metaphors in the other can be combined without requiring the programmer to manually reconcile both worlds with each other. We presented a very basic set of tools, which are modified versions of tools already known from other Morphic implementations, yet demonstrating the possibilities that are opened up by this new implementation approach of Morphic. We further presented an example development workflow in Transmorphic, and compared it to the alternative workflows possible in Squeak/Smalltalk and Lively Kernel. Here we saw, how we were able to interactively explore and adapt the functionally defined scene graph and evolve the functional description both programmatically and through direct manipulation interchangeably. We also indicated the areas of Transmorphic, that need improvement such as the reconciliation process itself, where besides *correctness* the preservation of the quality and intent within the symbolic description is of great importance. Furthermore in some scenarios, Transmorphic *requires* support from the programmer to successfully infer entity identity making the current reconciliation approach not as transparent, as one would like it to be.

There is still a lot to be discovered with regards to spreading the Transmorphic approach to other domains, that transcend the visual and spacial domain. Between the poles of functional and imperative programming concepts, a wide range of approaches to aid, support or guide an beginning, intermediate or professional developer can be found. In that sense the combination of imperative and functional world as demonstrated in Transmorphic, is just scratching the surface of the potential that lies in connecting and leveraging the synergetic effects between both worlds.

References

[1] P. Bagwell. *Ideal hash trees*. Technical report. 2001.

[2] A. Bohannon, B. C. Pierce, and J. A. Vaughan. "Relational lenses: a language for updatable views". In: *Proceedings of the ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM. 2006, pages 338–347.

[3] S. Burckhardt, M. Fahndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato. "It's alive! continuous feedback in UI programming". In: *ACM SIGPLAN Notices*. Volume 48. 6. ACM. 2013, pages 95–104.

[4] C. Calmez, H. Hesse, B. Siegmund, S. Stamm, A. Thomschke, R. Hirschfeld, D. Ingalls, and J. Lincke. *Explorative authoring of Active Web content in a mobile environment*. Technical report. 2013.

[5] C. Chambers and D. Ungar. "Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language". In: *ACM SIGPLAN Notices*. Volume 24. 7. ACM. 1989, pages 146–160.

[6] R. Chugh. "Prodirect Manipulation: Bidirectional Programming for the Masses". In: *arXiv preprint arXiv:1510.06788* (2015).

[7] R. Chugh, B. Hempel, M. Spradlin, and J. Albers. "Programmatic and Direct Manipulation, Together at Last". In: *arXiv preprint arXiv:1507.02988* (2015).

[8] E. Czaplicki. "Elm: Concurrent FRP for Functional GUIs". 2012.

[9] C. Elliott and P. Hudak. "Functional reactive animation". In: *ACM SIGPLAN Notices*. Volume 32. 8. ACM. 1997, pages 263–273.

[10] W. Engel. *Programming Vertex and Pixel Shaders*. Charles River Media, Inc., 2004.

[11] Facebook. *React*. 2013. URL: https://facebook.github.io/react/ (last accessed 2016-04-14).

[12] T. Felgentreff, J. Lincke, R. Hirschfeld, and L. Thamsen. "Lively groups: shared behavior in a world of objects without classes or prototypes". In: *Proceedings of the Future Programming Workshop (FPW) 2015*. ACM. 2015, pages 15–22.

[13] A. J. Goldberg. *SMALLTALK-80: the interactive programming environment*. Addison-Wesley, 1984.

[14] D. Goodman. *Dynamic HTML: The Definitive Reference: A Comprehensive Resource for HTML, CSS, DOM & JavaScript*. O'Reilly Media, Inc., 2002.

[15] R. Hickey. *Runtime Polymorphism*. 2009. URL: http://clojure.org/about/runtime_polymorphism (last accessed 2016-04-14).

[16] R. Hickey. "The clojure programming language". In: *Proceedings of the symposium on Dynamic languages*. ACM. 2008, page 1.

[17] R. Hickey. *Values and Change: Clojure's approach to Identity and State*. 2009. URL: http://clojure.org/about/state (last accessed 2016-04-14).

[18] G. Inc. *Minimizing Browser Reflow*. 2015. URL: https://developers.google.com/speed/articles/reflow#guidelines (last accessed 2016-04-14).

[19] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. "Back to the future: the story of Squeak, a practical Smalltalk written in itself". In: *ACM SIGPLAN Notices*. Volume 32. 10. ACM. 1997, pages 318–326.

[20] A. Kay. *Squeak Etoys authoring & media*. 2005.

[21] B. W. Kernighan, D. M. Ritchie, and P. Ejeklint. *The C programming language*. Volume 2. Prentice-Hall Englewood Cliffs, 1988.

[22] G. E. Krasner and S. T. Pope. "A description of the model-view-controller user interface paradigm in the smalltalk-80 system". In: *Journal of object oriented programming* 1.3 (1988), pages 26–49.

[23] J. Lincke, R. Krahn, D. Ingalls, M. Röder, and R. Hirschfeld. "The Lively PartsBin–A Cloud-Based Repository for Collaborative Development of Active Web Content". In: *System Science (HICSS), 2012 45th Hawaii International Conference on*. IEEE. 2012, pages 693–701.

[24] J. H. Maloney and R. B. Smith. "Directness and liveness in the morphic user interface construction environment". In: *Proceedings of the ACM symposium on User interface and software technology*. ACM. 1995, pages 21–28.

[25] J. McCarthy. "Recursive functions of symbolic expressions and their computation by machine, Part I". In: *Communications of the ACM* 3.4 (1960), pages 184–195.

[26] M. McGranaghan. "Clojurescript: Functional programming for javascript platforms". In: *IEEE Internet Computing* 6 (2011), pages 97–102.

[27] E. Meijer, M. Fokkinga, and R. Paterson. "Functional programming with bananas, lenses, envelopes and barbed wire". In: *Functional Programming Languages and Computer Architecture*. Springer. 1991, pages 124–144.

[28] Microsoft. *Retained Mode Versus Immediate Mode*. 2010. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ff684178(v=vs.85).aspx (last accessed 2016-04-15).

[29] S. M. Paramedics. *An introduction to deep code-walking macros with Clojure*. 2013. URL: http://blog.fogus.me/2013/07/17/an-introduction-to-deep-code-walking-macros-with-clojure/ (last accessed 2016-03-18).

[30] T. Schachmann. *Apparatus*. 2015. URL: http://aprt.us/ (last accessed 2016-03-18).

[31] B. Shneiderman. "Direct manipulation: A step beyond programming languages". In: *ACM SIGSOC Bulletin*. Volume 13. 2–3. ACM. 1981, page 143.

[32] D. A. Smith, A. Kay, A. Raab, and D. P. Reed. "Croquet-a collaboration system architecture". In: *Creating, Connecting and Collaborating Through Computing*. IEEE. 2003, pages 2–9.

[33] A. Taivalsaari, T. Mikkonen, D. Ingalls, and K. Palacz. *Web browser as an application platform: The lively kernel experience*. 2008.

[34] T. Teitelbaum and T. Reps. "The Cornell program synthesizer: a syntax-directed programming environment". In: *Communications of the ACM* 24.9 (1981), pages 563–573.

[35] S. Thompson. *Haskell: the craft of functional programming*. Volume 3. Addison Wesley Reading, 1999.

[36] N. Tillmann, M. Moskal, J. de Halleux, and M. Fahndrich. "TouchDevelop: programming cloud-connected mobile devices via touchscreen". In: *Proceedings of the SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software (Onward!)* ACM. 2011, pages 49–60.

[37] L. Torvalds and J. Hamano. "Git: Fast version control system". In: *http://git-scm.com* (2010).

[38] G. Van Rossum. "Python Programming Language." In: *USENIX Annual Technical Conference*. Volume 41. 2007.

[39] P. Wadler. "Monads for functional programming". In: *Advanced Functional Programming*. Springer, 1995, pages 24–52.

[40] Z. Wan and P. Hudak. "Functional reactive programming from first principles". In: *ACM SIGPLAN Notices*. Volume 35. 5. ACM. 2000, pages 242–252.

A. Appendix

Listing A.1: Helper functions, required for the clock to work

```
(def PI js/Math.PI)
(defn angle-for-hour [hour]
    (* (+ -0.25 (/ hour 12)) PI 2))

(defn angle-for-minute [min]
    (* (+ -0.25 (/ (self :seconds) 60)) 2 PI))

(defn point-from-polar [radius angle]
    {:x (* radius (.cos js/Math angle))
     :y (* radius (.sin js/Math angle))})
```

Listing A.2: Hour label component, used in the clock's implementation

```
(defcomponent hour-label
 IRender
 (render [self props _]
       (text
        {:id (str (props :label) "h")
         :position (point-from-polar
                 (* (props :radius) .8)
                 (angle-for-hour (props :hour)))
         :text-string (props :label)
         :font-family "Arial"
         :allow-input false})))
```

Listing A.3: The final definition of the clock component

```
(defcomponent clock
 IRender
 (render
  [{:as self, :keys [local-state]} {:keys [id extent position]} _]
  (let [{:keys [time]} local-state
       radius (/ (extent :x) 2)
       {:keys [x y]} extent
       ext (if (> x y) {:y x, :x x} {:y y, :x y})]
    (ellipse
     {:id id,
      :position position,
      :extent extent,
      :step
```

```
      (fn [_]
        (rerender! self {:time (get-current-time)})
        (refresh-scene!)),
      :border-width 4,
      :border-color "darkgrey",
      :fill
      "-webkit-gradient(radial, 50% 50%, 0, 50% 50%, 250,
        from(rgb(255, 255, 255)), to(rgb(224, 224, 224)))",
      :pivot-point {:x 0, :y 0}}
     (map
      (fn [hour]
       (hour-label
        {:id (str hour "h"),
         :label hour,
         :hour hour,
         :radius radius,
         :extent {:x 30 :y 30}
         :font-size 12}))
      (range 1 13))
     (polygon {:id "seconds"
               :fill "red"
               :rotation
               :vertices [{:x -10 :y 0} {:x 0 :y 0} {:x -5 :y 30}]})
     (polygon {:id "minutes"
               :fill "darkblue"
               :rotation (angle-for-minute (self :minutes))
               :vertices [{:x -10 :y 0} {:x 0 :y 0} {:x -5 :y 30}]})
     (polygon {:id "hours"
               :fill "darkblue"
               :rotation (angle-for-hour (self :hours))
               :vertices [{:x -10 :y 0} {:x 0 :y 0} {:x -5 :y 30}]}))))
  ↪ )))
```

Listing A.4: Overview of the clock morph's implementation in Squeak/Smalltalk

```
EllipseMorph subclass: #TiClock
 instanceVariableNames: 'secondHand minuteHand hourHand'
 classVariableNames: ''
 poolDictionaries: ''
 category: 'Clock'

TiClock>>handVertices

 ↑{5@130 . 10@0 . 0@0}

TiClock>>step
  | time |
   time := Time now.
   secondHand rotationDegrees: (time seconds * 6) + 180.
   minuteHand rotationDegrees: (time minutes * 6) + 180.
   hourHand rotationDegrees: (time hours * 30) + 180.
```

Listing A.5: Initialize method of TiClock

```
TiClock>>initialize
 | label |
 super initialize.

 self color: Color white darker;
     extent: 300@300.

 minuteHand := PolygonMorph
      vertices: self handVertices
      color: Color blue darker darker
      borderWidth: 0
      borderColor: nil.
 minuteHand rotationCenter: 0.05@0.05;
                position: 0@0;
   rotationDegrees: 180.
 self addMorph: minuteHand.

 hourHand := PolygonMorph
      vertices: self handVertices
      color: Color blue darker darker
      borderWidth: 0
      borderColor: nil.
 hourHand rotationCenter: 0.05@0.05;
        position: 0@0;
        rotationDegrees: 180.
 self addMorph: hourHand.

 secondHand := PolygonMorph
      vertices: self handVertices
      color: Color red
      borderWidth: 0
      borderColor: nil.
 secondHand rotationCenter: 0.05@0.05;
      position: 0@0;
     rotationDegrees: 180.
 self addMorph: secondHand.

 1 to: 12 do: [:hour |
   label := TextMorph new
             contents: hour asString;
             position: (Point r: 130 degrees: (hour * 30) - 90).
     self addMorph: label].

  self shiftSubmorphsBy: 145@145.
```

Listing A.6: Example of a complete reconciler that is extracted during macro expansion

```
{0 {:reification
```

```
      (fn [{:keys [m_1]}]
        `(IRender
           (render [self props submorphs]
             ~m_1))),
  :type :expr,
  :submorph-locations [1]},
1 {:reification
  (fn [self props submorphs]
    `(image
      ~(merge
       '{:extent {:x 200, :y 200},
         :position {:x 42, :y 42},
         :url "kermit.png",
         :on-mouse-enter (fn [e] (prn "Mouse entered!"))}
       props)
      submorphs)),
  :type :morph,
  :submorph-locations [2 3 5]},
2 {:reification
  (fn [self props submorphs]
    `(ellipse
      ~(merge
        '{:position {:x 0, :y 0},
          :fill "green",
          :extent {:x 100, :y 100}}
        props)
      submorphs)),
  :tag :morph,
  :submorph-locations []},
4 {:reification
  (fn [self props submorphs]
    `(text ~(merge
          '{:id (str i),
            :value "Hello World!",
            :position {:x (* i 10), :y (* i 10)}}
          props)
        submorphs)),
  :tag :morph,
  :submorph-locations []},
3 {:reification
  (fn [{:keys [m_4]}]
    `(map (fn [i]
          ~m_4)
        (range 10))),
  :type :expr,
  :submorph-locations [4]},
5 {:reification
  (fn [self props submorphs]
    `(clock ~(merge
              '{:id "Clock",
                :extent {:x 300, :y 300},
```

```
53                :position {:x 300, :y 300}} props)
54       submorphs)),
55    :type :component,
56    :submorph-locations []},
57 :active? false}
```

Aktuelle Technische Berichte des Hasso-Plattner-Instituts

Band	ISBN	Titel	Autoren / Redaktion
109	978-3-86956-386-2	**Software-Fehlerinjektion**	Lena Feinbube, Daniel Richter, Sebastian Gerstenberg, Patrick Siegler, Angelo Haller, Andreas Polze
108	978-3-86956-377-0	**Improving Hosted Continuous Integration Services**	Christopher Weyand, Jonas Chromik, Lennard Wolf, Steffen Kötte, Konstantin Haase, Tim Felgentreff, Jens Lincke, Robert Hirschfeld
107	978-3-86956-373-2	**Extending a dynamic programming language and runtime environment with access control**	Philipp Tessenow, Tim Felgentreff, Gilad Bracha, Robert Hirschfeld
106	978-3-86956-372-5	**On the Operationalization of Graph Queries with Generalized Discrimination Networks**	Thomas Beyhl, Dominique Blouin, Holger Giese, Leen Lambers
105	978-3-86956-360-2	**Proceedings of the Third HPI Cloud Symposium "Operating the Cloud" 2015**	Estee van der Walt, Jan Lindemann, Max Plauth, David Bartok (Hrsg.)
104	978-3-86956-355-8	**Tracing Algorithmic Primitives in RSqueak/VM**	Lars Wassermann, Tim Felgentreff, Tobias Pape, Carl Friedrich Bolz, Robert Hirschfeld
103	978-3-86956-348-0	**Babelsberg/RML : executable semantics and language testing with RML**	Tim Felgentreff, Robert Hirschfeld, Todd Millstein, Alan Borning
102	978-3-86956-347-3	**Proceedings of the Master Seminar on Event Processing Systems for Business Process Management Systems**	Anne Baumgraß, Andreas Meyer, Mathias Weske (Hrsg.)
101	978-3-86956-346-6	**Exploratory Authoring of Interactive Content in a Live Environment**	Philipp Otto, Jaqueline Pollak, Daniel Werner, Felix Wolff, Bastian Steinert, Lauritz Thamsen, Macel Taeumel, Jens Lincke, Robert Krahn, Daniel H. H. Ingalls, Robert Hirschfeld

ISBN 978-3-86956-387-9
ISSN 1613-5652