ReArch: A Reactive Approach to Application Architecture Supporting Side Effects

by

Gregory Conrad

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

December 2023

APPROVED:

Professor George Heineman

Professor Joshua Cuneo

***Abstract*— Modern application development encompasses a wide range of areas, spanning from high-level user-interface markup to low-level server and systems programming. These applications employ various multiparadigm techniques, predominantly drawing from architectural patterns rooted in Object-Oriented Programming and to a lesser extent, Functional Programming. Although several architectural patterns emerge, they often fall under just one Software Engineering subdiscipline, including State Management (SM), Component-Based Software Engineering (CBSE), and Incremental Computation (IC). However, these subdisciplines often remain isolated from each other, resulting in applications that rely on software libraries and frameworks that adhere to a singular approach. We introduce ReArch, a reactive and highly functional approach for designing an application architecture that provides a solution to SM, CBSE, and IC. ReArch includes novel innovations that support the composition of application code and arbitrary side effects, all while keeping application code functionally pure. ReArch has already been used to develop multiple production-grade applications across a variety of application domains, demonstrating its wide applicability. Going forward, ReArch will continue to provide applications with loose code coupling, maintainability, and testability by design.**

Contents

1 Introduction

This thesis aims to introduce an entirely new way to architect and build applications: **ReArch**. ReArch is a language-agnostic idea with two accompanying library implementations that provides a solution for a number of problem domains, including *state management*, *incremental computation*, and *component-based software engineering*. The solution is highly functional and declarative, while still supporting side effects through a novel realization.

The original motivation for ReArch was to provide a new state management framework for Flutter (Google's User Interface framework for building cross-platform applications) applications. Once a working prototype for ReArch was completed, it became clear that ReArch applies to much more than state management and can be used in a number of different problem domains. To enable this flexibility, ReArch acts upon two key observations to model applications:

1. The User Interface (UI) is a function of application state/data and any side effect(s). I.e., when given application state as an input, one can create a UI portraying that state for that given instant in time.
2. Application state/data is a function of other state/data and any side effect(s).

Accordingly, ReArch allows developers to simply define functions of application state/data and side effects for creating both state and UI, and in doing so improves the scalability of these applications.

Interestingly, through ReArch's novel approach to side effects, *almost all application code remains functionally pure, despite being able to handle arbitrary side effects*. Further, code is loosely coupled by design and testability is significantly improved, eliminating the need for any sort of complicated mocking. This outcome differs from other solutions, wherein side effects are often hard to manage due to their lack of testability and predictability as codebases change over time.

The original idea for ReArch (although dozens of design iterations ago) was inspired by a number of Flutter libraries, including Riverpod, flutter_hooks, and functional_widget. ReArch would not have been possible if not for these stellar, role-model projects. Throughout its numerous design iterations, ReArch borrowed many ideas from these packages along the way.

1.1 Motivation

A driving force behind ReArch's initial development was a feeling of unsatisfaction; Flutter remains intentionally unopinionated, forcing developers to manage state in a way of their choosing. When making a larger application, using Flutter's core components proves to take a lot of time and boilerplate, which many state management frameworks were created to solve. How-

ever, one quickly realizes that state management is more than just view/data management–it also encompasses data persistance, network requests, and much more to make a full application.

1.1.1 Solution Complexity

With new state management frameworks come new ways to handle these more complex scenarios; however, most frameworks tend to just provide a solution for each individual requirement (persistence, network requests, etc.) without creating a *general* solution that can apply to any sort of complex requirement. When looked at under a certain theoretical lens, this exact issue happens to be side effect composition (i.e., how can we combine different side effects together to make new and interesting side effects?).

1.1.2 OOP vs. FP Paradigms

To solve general application development problems, many patterns and paradigms have been created with Object-Oriented Programming in mind over the past several decades. While these solutions are abundant and tried and true [1], not as many paradigms exist for programs built with a function style in mind. Thus, when given a desire to implement an application in a functional manner, one might ponder how to create an application in a data-driven, reactive, declarative way without relying upon inheritance, as is often done in OOP applications.

1.2 Background

As mentioned above, ReArch provides a solution for numerous disciplines in software engineering, including state management, incremental computation, and component-based software engineering. Further, ReArch is applicable across programming languages and can be used for a multitude of different application requirements.

1.2.1 State Management

Originally, all GUI application development was done *natively*; i.e., using the tools and resources provided by the operating system to build out an application directly. As application development has evolved over the years, many new approaches to application development have emerged, along with differing approaches on how to build out actual applications, piece by piece.

A common requirement in GUI applications is to update a user interface when changes are made to application state. Although this requirement is simple at the surface level, it is quite open-ended and does not prescribe any specific means of achieving the goal, which leads to many different solutions. At a fundamental level, applications have an underlying model layer that contains the application state that is then transformed and rendered to the user. *State management frameworks* provide the ability to manage application state throughout the lifecycle of an

application, and often provide many other conveniences, such as data persistence and rendering optimizations. Examples of such state management frameworks can be found in the Related Works below.

To aid in developing user-facing applications, several design patterns have emerged over the years. Model View Controller (MVC) is one the most well known patterns and revolves around the separation of application data (model), user interfaces (view), and the application logic that bridges the model and view (controller) [2]. Due to bloat that often ends up in the controller layer in MVC, later came the Model View ViewModel (MVVM) pattern, derived from MVC, which employs *data-binding* to enforce a stark separation of concerns [3]. These patterns, in addition to many others, are often accompanied by techniques like dependency injection [4] and the Observer pattern [5] to create a fully featured application.

Employed to build graphical cross-platform applications, Flutter is a modern UI framework that uses the concept of *widgets*. Widgets are organized into a tree-like structure and are *built* to create a UI that is shown to the user [6]. Widgets are often UI code, but can also serve to handle state, accessibility/semantic information, and other important requirements in applications. Flutter is agnostic with regard to state management, allowing developers to choose a solution best suited toward their applications.

1.2.2 Incremental Computation

Incremental computation is a technique in which computation fragments are cached as the input data changes in an effort to save processing power. A common example is a spreadsheet, wherein cell changes only propagate to the affected cells which are then recalculated (versus refreshing the entire spreadsheet). This technique is used in numerous problem domains, including code compilation tools such as the Rust analyzer (which uses salsa) to only recompile code when it changes [7].

Incremental computation is often achieved by forming a data flow graph between data inputs and computations (which both serve as nodes). Then, all one has to do is compute the transitive closure starting from the changed node to determine all other nodes which need to be recomputed. An algorithm often employed for this task is topological sort, which acts similar to a depth first search with some key modifications to preserve computation ordering.

1.2.3 Component-Based Software Engineering

Component-based software engineering is a subdiscipline of software engineering that revolves around the notion of *components*, which are individual, self-contained building blocks a devel-

oper can *assemble* together to create an application. Software engineering researchers have long promoted the effectiveness of component-based software engineering for improving the code quality of software systems [8].

Component-based software engineering suggests a world in which components model reusable individual application features and functionalities, all composed together to create a full application. This model works well for purely functional software components (e.g., a spell checker), but cannot readily handle some components that are not purely functional (i.e., how would one component share some mutable data to several other components while maintaining consistency?). Additionally, components often relied heavily on external dependencies, such as a database management system, and their internal implementation directly accessed persistent storage (e.g., through SQL queries). Most of the existing component implementations simply avoided the real problems that are evident when the interaction between two components exposes potential side effects.

2 Related Works

Many works have been published in the last couple decades in regards to state management, incremental computation, and component-based software engineering.

2.1 State Management

State management has roots in some of the first applications created and has evolved significantly with the introduction of patterns like MVC and MVVM. Many techniques and methodologies exist today for application development, with the flow and order of influence of some of these approaches depicted in Figure 1. ReArch, although largely introducing novel approaches to application development, still drew inspiration from a vast array of techniques.

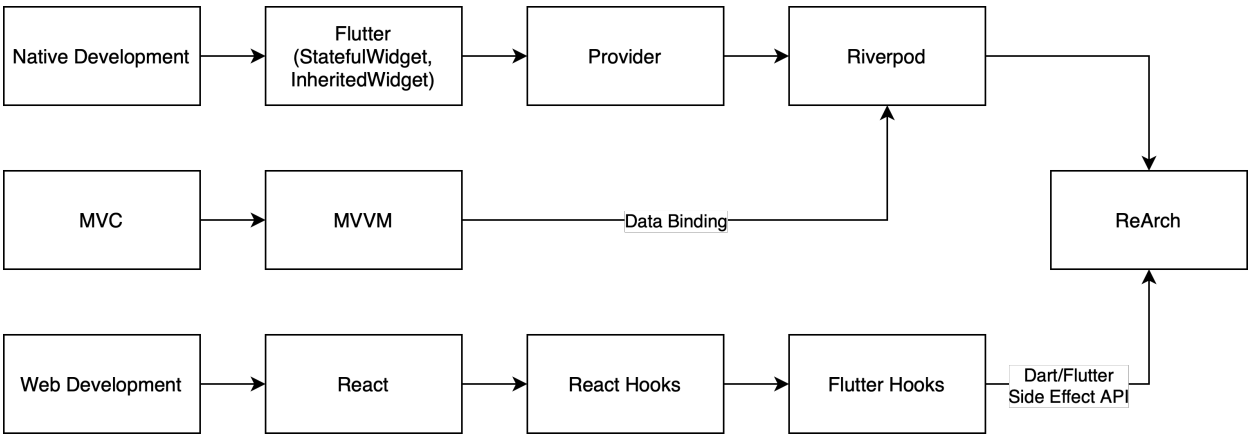

Figure 1: The order of influence for various application-building techniques, especially with regards to state management.

2.1.1 Flutter

Flutter has simple builtin support for state management through the provided *StatefulWidget* and *InheritedWidget*. While a StatefulWidget maintains some piece(s) of changing state throughout its lifecycle, an InheritedWidget provides some value to its descendants in the widget tree. While the state of a complex application can be built upon just these two building blocks, a developer quickly realizes that StatefulWidgets and InheritedWidgets both require substantial boilerplate, in addition to minimal code reusability for common operations.

2.1.2 Provider

To solve the boilerplate of InheritedWidgets, the library *Provider* was designed to give developers an avenue to provide application state to graphical widgets via *providers* [9]. Providers simply provide some typed data to their children in the Widget tree, but the pattern as-is was shown to have numerous limitations. Provider is not compile-time safe, instead opting for runtime checks. Further, one must create a new type (often an entire class) since one cannot have multiple providers of the same type of data. Side effects, such as asychronous code, can be tricky to await and properly handle in UI since there isn't as much support for such side effects provided in provider.

2.1.3 Riverpod

Thus, after some years, it became clear that Provider had some fundamental underlying limitations, and the same author went on to fix said problems in *Riverpod*. Riverpod is self-described as a "Reactive Caching and Data-binding Framework," and serves as a turnkey state management framework [10]. However, Riverpod has some fundamental weaknesses due to its complicated design that originally inspired the creation of ReArch.

2.1.3.A Many Types

Riverpod has a large number of different types of components that take awhile to fully grasp. Code generation was introduced in the second major version to help with this problem, but there are still a mutlitude of different types under the hood that have to account for all one-off scenarios and users sometimes have to pick between different options which can also confuse beginners.

2.1.3.B Provider Functionality

Like Provider, Riverpod also exposes *providers* to house application state; however, Riverpod handles providers differently. Providers in Riverpod can be of any type and can have inter-dependencies, forming a dataflow graph. While this approach works well on paper, two problems are quickly uncovered when building an application:

1. How does one acquire application state that relies upon some keyed data?
2. How does one ensure the graph doesn't constantly grow in size when some subgraphs may no longer be in use?

To solve the first question, Riverpod introduced *Family* providers, and for the second, *AutoDispose* providers. However, the manner in which these two features were introduced often results in overcomplication and bad practices.

2.1.3.B.I Family Providers

Families have two underlying implementation problems. The first is that family providers internally act as if a new provider instance is created for each unique set of inputs. This in turn causes the globally-stored cache to grow substantially, and *requires* AutoDispose to prevent leaks (but AutoDipose itself has problems). The second problem is that families are globally scoped, despite only ever being required in a locally scoped context. To best explain that point, one must consider where family providers can be used. There are two possible answers:

1. When the provider is used locally, the local key and state will be promoted to global state (where family providers reside). This violates best practices of minimizing the scope of state and variables.
2. When the provider is used globally, one would need to store the family key in one provider, and then use the family provider to turn those arguments into a new provider accessible globally. While family *can* work for this situation, it doesn't entirely make sense over normal provider composition. To prevent leaks, one is forced to make the provider AutoDis-

pose. Further, the framework must deal with the complexity of caching the different in-use versions of the family, which then forces the parameters to override `hashCode` and `==`.

Both of the above situations are an ideal scenario for the factory pattern, borrowed from Object-Oriented Programming. The factory pattern provides a means to create some object or application state via some non-injectable data/key, which is exactly what families were introduced to solve.

2.1.3.B.II AutoDispose Providers

While being able to clear a cache when its no longer needed is generally a good idea, implementations of said idea can be challenging to do effectively. Riverpod's implementation of AutoDispose providers has a few limitations. All providers must be explicity declared as AutoDispose or not, which can easily cause maintenance bugs as applications grow in size and complexity when one expects a provider to have previous data when in fact it was disposed (or vice versa). Further, timer-based disposal delays are exposed (`disposeDelay`) in order to prevent disposal of certain AutoDispose providers when navigating around a Flutter application. Relying on something such as timing in order to prevent bugs is often considered a bad practice when a true fix is more appropriate. Finally, the existing implementation requires slew of types throughout the framework (or the use of a IDE linter to point out problematic code), in addition to a tight coupling between the main package and the auxillary Flutter support package to provide the necessary AutoDispose functionality. Application state is divided between two categories–global state and ephemeral (disposable local) state. If some state *needs* disposal, there is a high likely hood that it is in fact ephemeral state. In odd cases where there is a large global cache (which often is stateful for an application), it often should be handled on case-by-case scenarios due to differing data retention requirements between applications.

Similar to family providers, AutoDispose providers are adequately addressed via the factory pattern. Factories can be used to manage an object's lifecycle as an application requires. Also, ReArch introduces *automatic* garbage collection that identifies capsule subgraphs which can be safely disposed without any user intervention.

2.1.3.C Expressiveness

Despite often favoring declarative approaches when composing providers, this approach clashes with some procedural approaches throughout the framework. Notifiers are a construct introduced in provider that aim to manage *mutating* state via a class that exposes mutating methods. However, the state is a field of the class and requires explicit assignment to mutate, in addition to

a *build* method that rebuilds the data. Beginners often get confused with this distinction, especially when mutations (moreso with asynchronous code) require procedural approaches. In addition to the `state`, Notifiers contain several other fields and intricacies like `future` that are not declarative and especially confused to newcomers. Notifiers were design to solve the problem of mutating state over time through object-oriented programming, but the approach doesn't mix as seamlessly with the functional/declarative/reactive approach found elsewhere in the framework.

2.1.3.D Scoping State

Riverpod is specifically designed for global application state, but provides ways to be used for local state. However, there is no *easy* way to scope state to a particular Widget, and when you do, requires advanced features that are error-prone, including having to manually specify a provider's dependencies (when they are already implicitly declared in the provider itself). Advanced features like scoping state are inherently complex and hard to grasp, and an application will likely need some of them as it furthers along in development.

2.1.3.E Lack of Extensibility

The core framework has a lot of necessary functionality builtin, but doesn't expose a way to easily *extend* that builtin functionality. Several crucial features, like data persistence and watching mutations, have been open issues for some years. If the framework provided a way for third parties to easily build this functionality, then this would be a non-issue; however, features like these require a deep integration with Riverpod as-is and have yet to be implemented.

2.1.4 BLoC

Business Logic Component (BLoC) is a design pattern that has a companion library for Dart/Flutter. BLoC revolves around the idea that the UI can emit events (e.g., a button press or text input) which are then picked up by an intermediate layer that emits new states for the UI to display. The middle layer thus can be expressed as a function consuming the current state and the fired event, returning the new state. While this pattern effectively removes coupling between the UI and state, it often results in overly verbose code (developers need to create boilerplate for every single possible event and state) and causes the accompanying package to include "cubits" for simpler state interactions. As such, the BLoC pattern is a great solution for complex UIs with many possible non-intersecting states, but for the vast majority of scenarios it is more verbose than needed to acheive predictability. Further, the BLoC pattern itself is easy to acheive without the companion Dart/Flutter library; Hooks and ReArch both provide a reducer side effect that enable the BLoC pattern.

2.1.5 ReactiveX

On a different trajectory, ReactiveX is a language agnostic methodology that extends the Observer pattern to enable functional code similar to that of the Iterator pattern [5]. ReactiveX and the observable pattern are often touted as an easy way to approach application modeling using side effects and simple transformations on that data. ReactiveX is widely used across the industry, especially in frontend code with RxJS. In Dart/Flutter, `Streams` are often used to mimic the same pattern.

Despite wide adoption, ReactiveX suffers from a fundamental limitation when building larger applications: the Observer pattern cannot *correctly* support side effects for composite observables due to the lack of a dependency graph (side effects may be incorrectly triggered or be left in an invalid state when there are multiple observables that have dependencies on each other). While the side effects and transformations from ReactiveX are a useful approach to data modeling, it only works on individual observerables in ReactiveX and disallows proper composition. The lack of a mechanism to correctly support side effects thwarts any form of useful composition/assembly across observables, at least at scale.

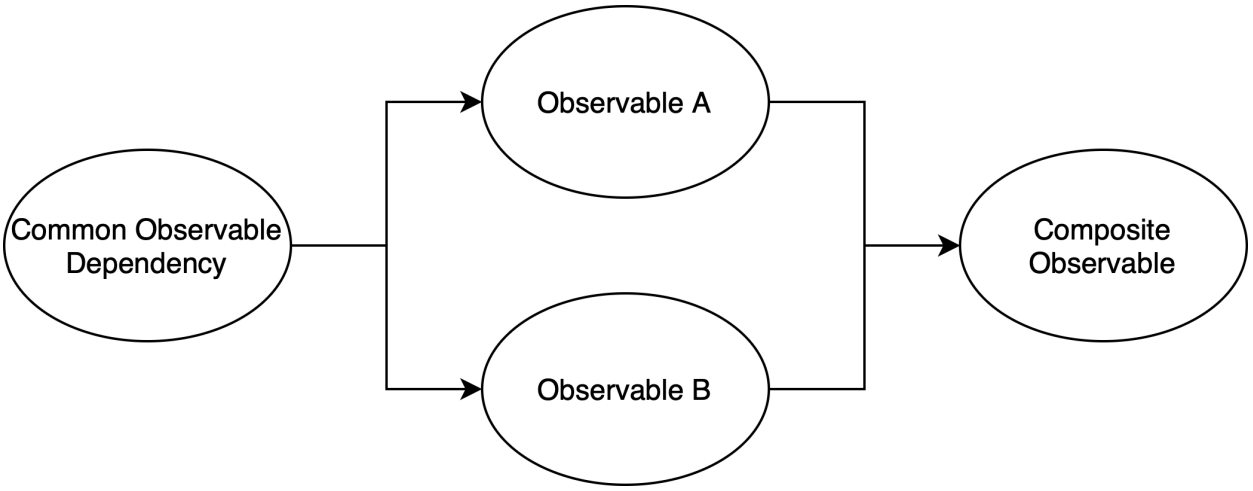

Figure 2: Sample set of composed observables.

For example, take Figure 2. If the common observable dependency were to update, it'd propagate changes directly to Observable A and Observable B. Before yielding control flow to B, A would first propagate its own changes to the composite observable, which houses some side effect (for the sake of this example, logging). It would then log the new value of A, but the old value of B. However, right after the first log, B would be notified via the common dependency observable, causing another log from the composite observable with the actually correct value. As can be

seen here, updates without a proper notification order (caused by no dependency graph) result in incorrect triggering/state of side effects which is unacceptable for the matter of correctness.[1]

2.1.6 React Hooks

React Hooks was one source of inspiration for the original design of ReArch and still has much influence on the exposed API of the Dart implementation today. Hooks opens the door for *composing* side effects, which is an expressive technique enabling declarative side effect code. While they work well for their intended use, hooks suffer from several limitations. Hooks only work for managing local state and cannot be used outside of UI code. Further, hooks often require explicitly declaring the environment variables a closure closes-over in order to run side effects only on certain builds. Finally, while mocking is possible during unit testing, UI code still remains tightly coupled to the React library since hooks are builtin library functions that have side effects.

2.1.7 Dependency Injection

Dependency Injection (DI) is a common practice through frameworks like Guice and Dagger, and are widely used across the industry. DI is often used to maintain application-wide state via an injected singleton state object, which then enables state sharing. However, existing solutions rely upon runtime introspection (Guice) or code generation (Dagger) which impact runtime and compiletime speeds, respectively. Further, while injecting singleton objects works adequately for simple mutations and side effects, complex state objects suffer from the lack of easy observability when using dependency injection. If the observer pattern is used to mitigate these issues, the same exact issues discussed with ReactiveX above are now prevelant. ReArch also serves as a means to achieve proper dependency inversion, but does so in a functional and reactive manner.

2.2 Incremental Computation

Numerous works have been published to optimize prior solutions in incremental computation. When spreadsheet programs were first introduced decades ago, computer scientists quickly realized they would not be able to update an entire spreadsheet each time data changes (especially on the resource-constrained devices of the time). Around the same time, several initial approaches to incremental computation were born to accomodate similar situations. In addition to spreadsheet applications, incremental computation also has a strong foothold in other spaces, such as to aid in code compilation [7].

[1] Astute readers familiar with ReactiveX may point out that `zip` can be used to solve the provided problem in Figure 2. However, try adding in another dependency for the composite observable that that does not depend upon the common observable– it is quickly evident that it is not feasible to correctly compose observables in this manner.

2.2.1 Adapton

Many incremental computation implementations were inspired by Adapton. Adapton is a *demand-driven* approach to incremental computation, providing significant performance improvements over previous incremental computation implementations by skipping computations that have no demand [11]. While Adapton is especially well-suited toward particular workloads, it suffers from a complicated model that cannot support arbitrary side effects; the lack of side effects is a common limitation in incremental computation approaches that prevent adoption in many more applications. While many incremental computation frameworks support mutating inputs [11], a manually mutated input cannot exhaustively handle all forms of side effects. While such a framework handles computations effectively, one cannot make a full application using only computations (without side effects) outside of a few select disciplines.

2.3 Component-Based Software Engineering

Much academic work has been published in the past couple decades in regards to component-based software engineering. As time passed, novel component models were introduced that aimed to solve unique challenges faced by other component models.

2.3.1 Exogenous Connectors

In an attempt to separate the design and deployment phases of component-based software engineering and to properly define how components operate [12], Kung-Kiu Lau created a component model called *Exogenous Connectors* that features atomic components (representing purely functional computations) and composite components (which orchestrate the activities of atomic and other composite components) [13]. However, components have a number of competing requirements, such as mutating state, storing data persistently, *in addition* to sharing data with other components; consequently, component composition is difficult in practice for a full application that has to handle impurities. The underlying problem of this observation is in part due to the handling of side effects–if a component model was to natively support side effects, then arbitrarily complex components could easily be composed together as expected due a the uniform API.

3 Design

The original inspiration for ReArch came after uncovering pain points in many exisiting solutions. While exisiting solutions all brought different features to the table, there wasn't one solution that was easily able to accomodate all of a large application's requirements. Thus, ReArch was created to reach feature parity with the many existing solutions by focusing on *extensibility*

while also incorporating novel innovations. To complement extensibility, the biggest overarching design goal is *composability*. It is often easiest to model applications simply by their data and its flow, which composability enables with loosely-coupled components.

3.1 Goals

1. Small API Footprint with Extensibility

In an age with many established competing libraries and frameworks, simplicity is often overlooked. Maintaining a small core-API footprint is critical for ease of adoption. Once one learns the core components of a framework and how they interoperate, it is easy to look at more in-depth documentation and/or examples to build upon them. ReArch was not only designed to ease adoption through a small API, but also to facilitate future external contributions in order to drive a user-friendly ecosystem.

2. Handling Side Effects with Composition

Side effects, such as asynchronous operations and interfacing with external systems, can be challenging to correctly manage in software systems. ReArch introduces a new side effect model to the table that enables composition by handling side effects in a modular and predictable manner. With side effect composition, developers can leverage declarative code to reduce previously error-prone and unmaintainable aspects of their applications.

3. Reactivity through Declarative Code

By describing the desired state of a system rather than prescribing the steps to get there, declarative code is able to simplify development through increased code readability. Further, reactivity is paramount to many modern applications due to their reliance on external real-time data, user interactions, and other such side effects. ReArch aims to achieve reactivity through declarative code; through interpreting what is defined declaratively, a framework can undergo internal optimizations, all while powering reactivity through its job as the orchestrator. The same level of power is not easily replicable with standard imperative programming, at least without a high level of effort (abstractions make assumptions and optimizations easier, just as can be seen with programming languages compilers).

4. Applicability Across Application and Solution Domains

ReArch is designed with the intenion to be used across a wide range of application and solution domains by providing a consistent, highly adaptable core framework. With a solid and flexible

base, ReArch excels as a methodology in writting many different types of applications, ranging from mobile phone to realtime and data-intensive serverside applications.

3.2 Capsules

In ReArch, *capsules* are the fundamental unit of computation that encapsulate some data (thus their naming). Capsules are pure functions that return an *output*, which is an immutable copy of the capsule's data. The most primitive capsule, for example, is one that returns a constant value (see `count` in Listing 1). As every capsule is simply a function, capsules are uniquely identified by their function's signature. Capsules all must consume a `CapsuleHandle` and return their typed output, thus following the form of `Fn(CapsuleHandle) -> Output`. Note that capsules are commonly higher-order functions, returning functions as their output data, enabling some new patterns and paradigms.

```
int count(CapsuleHandle use) => 0;
int countPlusOne(CapsuleHandle use) => use(count) + 1;
```

Listing 1: A basic Dart example of how capsules enable declarative code; the `count` capsule simply provides a numerical count, and the `countPlusOne` capsule will always represent the current `count`, plus one.

3.2.1 Dependencies

A capsule can compose together the output of other capsules when computing its own output. As such, each capsule can have zero or more *dependent capsules*. When a capsule C_1 has a dependent capsule, C_2, then a change to the output of C_1 may cause a change to the output of C_2. This relationship is represented using notation $C_1 \longrightarrow C_2$ and we would say that the dependent capsules for C_1 is the set $\{C_2\}$. Inversely, one can also infer the dependencies for a capsule, C, because this would be the set of capsules, $\{C_i\}$, such that $C_i \longrightarrow C$. This relationship of capsule dependencies and dependents is demonstrated in Figure 3.

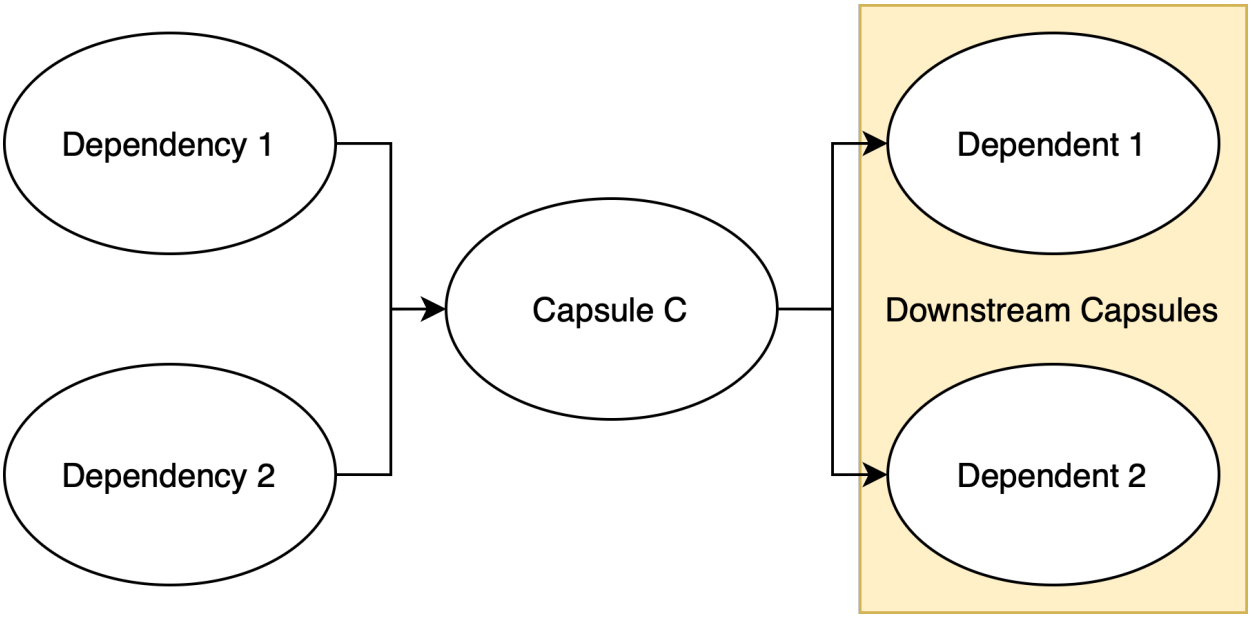

Figure 3: A capsule, C, with a set of labeled dependencies and dependents. The set of dependent capsules are often referred to as *downstream capsules*.

As shown in Figure 3, the highlighted region of the dependent capsules is labeled as the *downstream capsules*. The set of downstream capsules represeents the set of capsules that may need updating when a capsule C changes, and is calculated by the transitive closure of the dependency $\longrightarrow$ starting from C.

Dependency relationships are automatically inferred and kept up-to-date by the accompanying ReArch implementations; however, for illustrative purposes, take Listing 1. `count` has no dependencies, but one dependent (`countPlusOne`). On the opposite hand, `countPlusOne` has no dependents, but one dependency (`count`). Thus, their relationship can be described as `count` $\longrightarrow$ `countPlusOne`.

3.2.2 Side Effects

While capsules are always written as pure functions, the novel contribution of this thesis is to enable a capsule to have *side effects* (despite capsules' functional purity), which are modeled as pairing private mutable data D with a mechanism to mutate D. Note that D is only visible to the side effect itself. Any change to D (based on the provided mechanism) forces the capsule to *rebuild* (more on capsule life cycles later), and potentially emit a different output.

A capsule may or may not have side effects; capsules with no side effects are known as *idempotent* and enable a multitude of garbage collection strategies. Interestingly, the garbage collection of idempotent capsules is also what enables ReArch to be perfectly *demand-driven*, as defined in

Adapton. Capsule dependencies can easily change with time, allowing demand to be modified over the course of an application's run.

3.2.3 Definition

More formally, the state of a capsule, C, is defined by the tuple (*dependencies*, *side effects*), where *dependencies* is the set of capsules upon which the computation of C depends upon and *side effects* enumerates the side effects (if any) for C. This clean separation of dependencies and side effects ensures that capsules can be composed together even while supporting arbitrary state changes from side effects.

3.2.4 Life Cycle

When looked at as a state machine, capsules only have two distinct states, and several life cycle events that can transition between these two states. The first, initial state, is the capsule being not created/cached; the second state is when the capsule is created/cached.

To create/cache a capsule, a capsule must be *built*, which involves invoking the capsule (as a reminder, capsules are just pure functions) to get its output. The concept of *building* capsules is borrowed from other frameworks, like widgets in Flutter [6] and providers in Riverpod [10]. Building capsules enables declarative definitions, and let the underlying framework handle all of the imperative glue behind the scenes. The approach is similar to constructors in dependency injection as used in OOP, but is instead adapted to apply to a functional approach.

When a capsule is already cached, it can be rebuilt (either from a dependent rebuilding or from a side effect mutating). This causes the capsule to (possibly) emit a new output. If this output changes, a capsule's dependents must also rebuild; if not, dependent rebuilds may be skipped as an optimization.

Finally, capsules will be disposed. For the case of idempotent capsules, this often happens when a dependent is rebuilt and the idempotent capsules have no demand (i.e., is in an idempotent tree subgraph). Non-idempotent capsules will only ever be disposed when the container (discussed later) is disposed, as they are stateful and tied to the container's internal store directly.

This overarching flow can be seen in Figure 4.

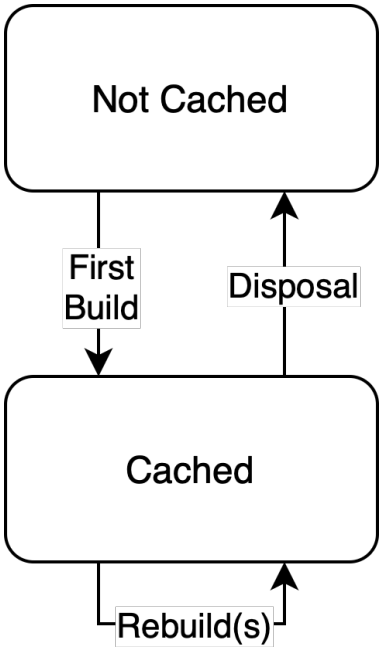

Figure 4: A capsule's life cycle of two states: either cached or not cached. Capsules can be built for a first time, experience rebuild(s), and eventually get disposed.

3.2.5 Composition

New capsules can be easily created via *composition* by referencing the output of some dependency capsules. Composition enables capsules to remain loosely coupled, despite depending upon other capsules' data. As an example, take three capsules where $A \longrightarrow C$ and $B \longrightarrow C$. In this instance, C is composed with the current data of A and B. Further, the underlying data of A and B may be easily swapped out in testing, as the `CapsuleHandle` merely serves as an interface (Dart) or a mockable structure (Rust).

3.2.5.A In Applications

Applications are assembled via the composition of independent capsules as shown in Figure 5. Thus, there will be possibly numerous backing capsules in a single application, where each individual, boiled down feature requirement tends to map one-to-one with a corresponding capsule. Through architecting an application via the lens of capsules, entire applications can be built with very loose coupling and rapid feature adoption (by simpling creating a new capsule for each new feature).

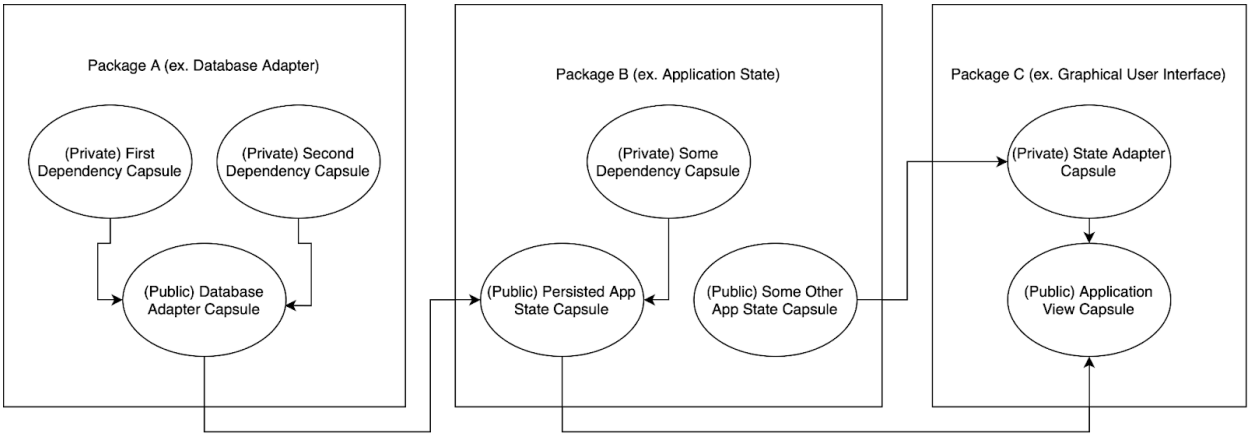

Figure 5: A sample application's capsule dependency graph.

3.2.6 Data Flow Graph

The composition of capsules internally forms a *data flow* graph. If given a collection of capsules developed with composition, $\{C_1, C_2, ..., C_n\}$, one can define the directed acyclic graph $G = (V, E)$ where V is a set of vertices and E is the set of edges. Each capsule C_i maps to vertex V_i, so there are n vertices in V. If C_i has a dependent capsule C_j then the edge $C_i \longrightarrow C_j$ exists in E. This resulting graph is called the *data flow* graph, as it represents the fashion in which updates to capsule data flow.

3.2.7 Dynamic Capsules

As traditional capsules are *static* in the sense that they are known at compile time, ReArch also supports the concept of *dynamic* capsules that can be determined at runtime. A common use case of dynamic capsules is for data that is keyed by some input, such as traditional incremental computation and dynamic programming problems. For an example, take Listing 2, which details how to evaluate the fibonacci sequence using capsules and ReArch's builtin caching mechanism.

```rust
struct FibonacciCapsule(u8);
impl Capsule for FibonacciCapsule {
    type Data = u128;

    fn build(&self, CapsuleHandle { mut get, .. }: CapsuleHandle) -> Self::Data {
        let Self(n) = self;
        match n {
            0 => 0,
            1 => 1,
            n => get(Self(n - 1)) + get(Self(n - 2)),
        }
    }

    fn eq(old: &Self::Data, new: &Self::Data) -> bool {
        old == new
    }

    fn key(&self) -> CapsuleKey {
        let Self(id) = self;
        id.to_le_bytes().as_ref().to_owned().into()
    }
}
```

Listing 2: An example of how to use dynamic capsules to evaluate the fibonacci sequence with the Rust ReArch implementation.

3.2.8 The `CapsuleHandle`

The CapsuleHandle is a composition of two types: one enables composition via the reading of other capsules' data (the *CapsuleReader*) and another gives capsules an API to interact with their registered side effect(s) (the *SideEffectRegistrar*).

3.2.9 The `CapsuleReader`

The CapsuleReader itself is a closure; when invoked it retrieves the data of the supplied argument capsule while internally updating the capsule's dependency relationships. In code, the CapsuleReader is `use(someCapsule)` in Dart and `get(some_capsule)` in Rust. Capsule cycles can be caught via the CapsuleReader, but both accompanying implementations will stack overflow (at the time of writing); a proper error message is a nice addition that may be added in the future.

While more than one capsule can cause dependency cycles, a capsule is allowed to perform *self reads* as an alternative to side effects. When creating the Rust implementation, a first class API

to access a capsule's previous value was attempted, only to find it was difficult to do correctly (at least not without a clunky API). Interestingly, this can perhaps be attributed to Rust catching a possible logic bug; in order for a capsule to read its previous value, it must inherently be non-idempotent, as relying upon past state in turn makes something non-idempotent. Thus, the chosen solution is to force capsules that may self read to register a side effect that checks whether or not they have already built before trying to read their own data, ensuring they are non-idempotent (as checking whether a capsule is in its first build is done via a side effect).

3.2.10 The `SideEffectRegistrar`

Capsules are intended to be functionally pure; i.e., a capsule itself may not use any code that has side effects directly. When a language allows for it, this can be partially enforced at the language level (e.g., Rust has an `Fn` and `FnMut`, where `Fn` will prevent the mutation of variables captured by the function). However, this is circumventable via interior mutability and synchronization primitives like mutexes, which is actually how some of ReArch's unit tests are implemented (to ensure capsules are built as expected). In application code, however, such techniques are considered a bad practice as they break the reactive nature of ReArch.

Further, erroneous bugs may occur in developer's code if side effects are used without the provided mechanism (the `SideEffectRegistrar`). The `SideEffectRegistrar` allows capsules to use side effects by giving capsules a look into the outside world (more on side effects in the following sections), by using a container as the middleman.

3.3 Containers

Because capsules are pure functions by design, they cannot contain any additional metadata or state that is required for an application. One needs an external mechanism to orchestrate capsule lifecycles and build capsules in their topological order; this mechanism is named the *Container*, as containers contain the state of a set of capsules. Containers are first created, can then be used to interact with a given set of capsules, and are finally destroyed (often at application exit), erasing the state of all contained capsules.

When created, a *container* has no capsules. Capsules can be added to the container one at a time, or as a set of capsules. An application is divided into numerous capsules, each of which performs some computation towards the greater application.

Containers are an effective approach with an analogous solution in several other modern state management solutions [10], [14] due to their ability to effectively remove code coupling in a

manner similar to dependency injection. Thus, with containers comes increased testability, easy implementation switching at runtime, and everything else expected from loosely coupled code.

Containers provide the core functionality to retrieve the output from a capsule. Other functionality, like reacting to capsule changes, can be accomplished through capsule composition and the creation of new capsules; however, some ReArch implementations may choose to expose such common methods on the container directly for ease of use or for container space optimizations. Further, each implementation may expose other container methods depending on the applications meant to be built with the given implementation, including the ability to manually create container transactions in Rust (more on transactions later).

While reading capsules is the only required element of the public-facing API, containers undergo numerous interal processes as well, including building capsules, rebuilding capsules, caching/preserving state, and a configurable automatic garbage collection.

3.3.1 Reading Capsules

Containers act lazily and only perform what is requested of them at the time of request. When an actor reads the output value from a capsule *C*, the container will lazily ensure that all dependencies for *C* are instantiated, instantiate the capsule, *C*, itself if needed, and then return the current output for *C*. An example of this can be seen in Figure 6, where an actor reads capsules A, B, and C, where B is a dependency of C. When C is read, the container first builds and caches B, followed by building, caching, and returning C.

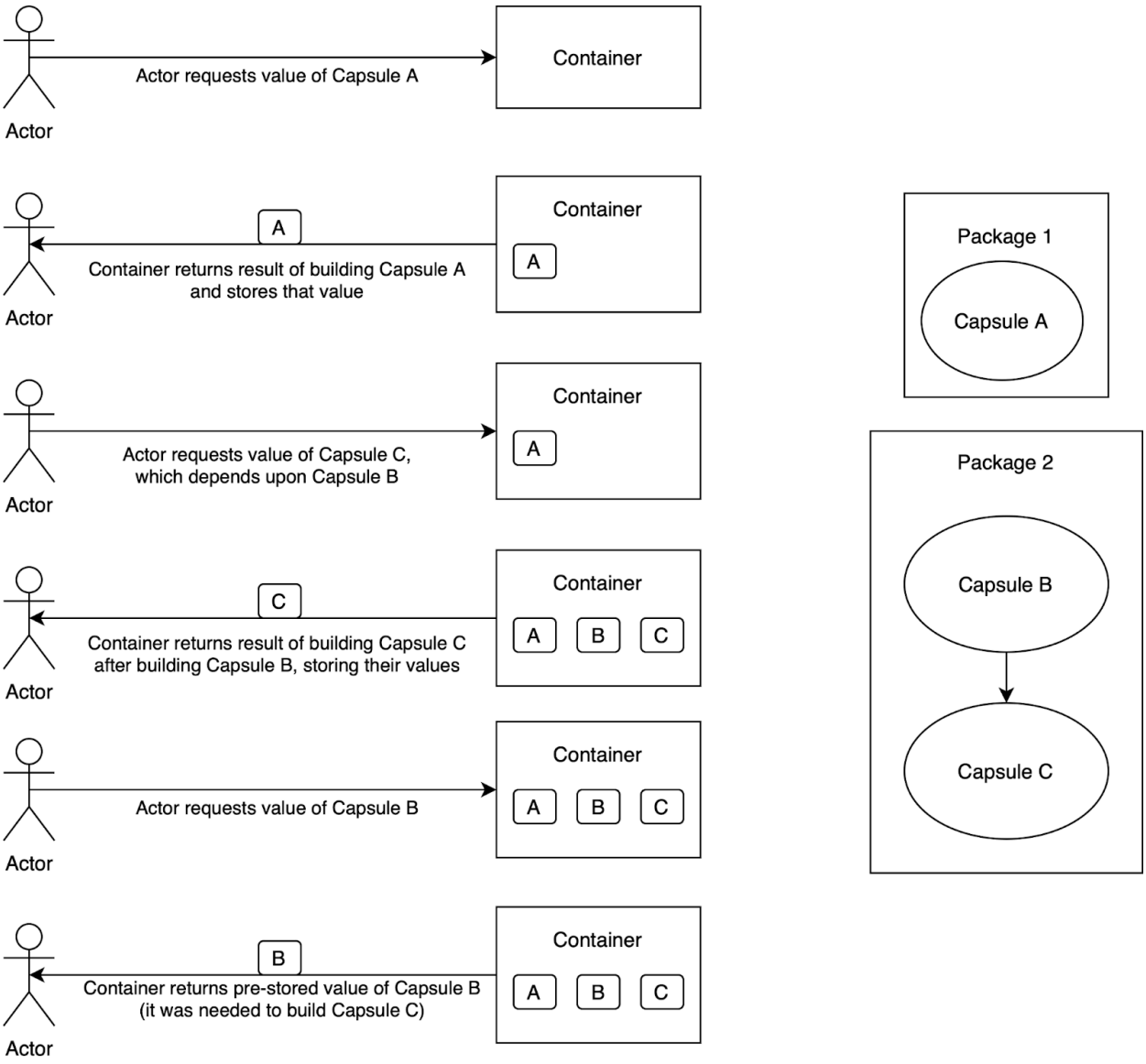

Figure 6: Timeline of container initializing three capsules: A, B, and C, where C has a dependency on B.

In order to enable this lazy execution of requests, containers undergo a particular process while reading capsules. A container first checks to see if it has already cached the output of the capsule. If it does, it simply returns that cached value; if it does not, the container must build the requested capsule and cache its output (doing the same for any dependencies along the way).

3.3.2 Building Capsules

To build capsules, the container creates a temporary lease of itself, which is exposed in the ReArch API as the previously discussed *CapsuleHandle*. This lease contains the current state of the container, giving capsules a copy of the current and *consistent* state of all their dependencies

(more on consistency later), in addition to the state of its registered side effect(s), if there are any. Further, as a capsule builds, the container forms/updates its internal dependency graph with the new dependency-relationship edges of the capsule in question, marking the capsule as a dependent of its dependencies (and the two accompanying implementations do the opposite as well to ease the implementation complexity, although *this is not necessarily required*). A container must refresh dependency relationships on every build because capsules able to dynamically change their dependencies by *conditionally requesting* their dependencies in *rebuilds* (i.e., in an if/else branches).

3.3.3 Rebuilding Capsules

When a capsule C rebuilds, either due to a side effect triggering a rebuild or due to a dependency rebuilding, the container is responsible for rebuilding all *downstream* capsules. As described earlier, the set of *downstream* capsules is computed from the transitive closure of C's dependents. These *downstream* capsules must be rebuilt in their topological ordering to ensure correctness, which is acheived by a topological sort over the graph using a modified depth-first search algorithm, starting from the rebuilding capsule. Once a container computes a suitable topological ordering, the container then starts building the capsules in order and can skip building certain capsules when all of the capsule's transitive dependencies don't change (i.e., emit the same data).

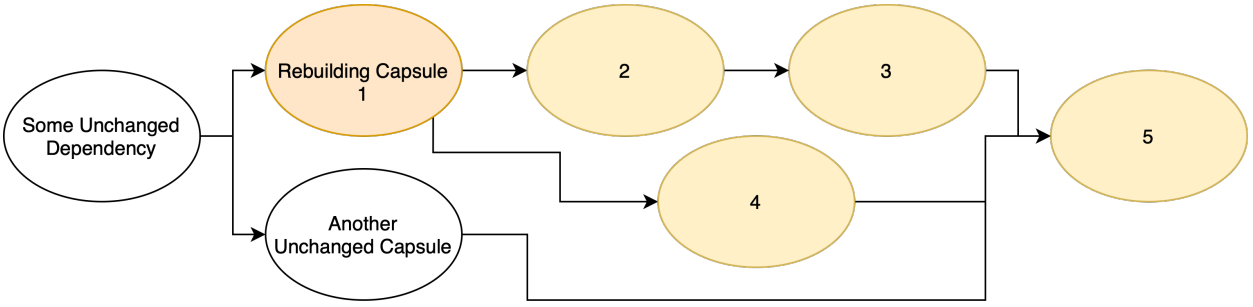

Figure 7: A completed topological ordering of a sample capsule dependency graph. Highlighted capsules are numbered in their build order.

3.3.4 Caching/Preserving State

A key requirement of containers is to cache and preserve the state of capsules when necessary. Reading capsules is a fairly common operation, and thus needs to be $O(1)$ to keep applications performant. Thus, a data structure with constant-time lookups is required.

To meet this requirement, a hashmap is the data structure of choice. The key observation here is that *capsule functions act as keys* in this hashmap, while the capsules' states serve as the values.

While a capsule's state must clearly include the capsule's current output, capsule state also includes several other pieces of information necessary for container operation. Capsule state also includes a capsule's private side effect data (if any), in addition to any dependent capsules in the event of a rebuild.

Interestingly, in the configuration outlined above, a container's hashmap acts similarly to a virtualized version of the heap that today's programming languages offer. A capsule's definition (the function) acts as a pointer of sorts, and the container acts as the heap that contains the data/memory for any given capsule/pointer. The analogy is further extended by the fact that all data is stored contingently in the heap despite the lexical scoping of any variables in a program, which is the same way capsules work (capsules' states are stored at the same level in the container, but capsules can only depend upon capsules available at their lexical scope). Modeling containers in this manner is necessary for a few different reasons, the biggest of which is to ensure that there will only ever exist one instance of a capsule even when multiple others depend on it (traditional hashmaps allow for only one value per key by design).

3.3.5 Concurrency

With the backing data structure of a hashmap, one must also question how containers handle multithreaded and concurrent workloads. In languages featuring an event loop, such as Dart and JavaScript, concurrency is a non-issue as any object can only be accessed by one actor at a time by design. However, in languages like Rust wherein concurrency is vital to application design, a proper concurrency model must be chosen.

3.3.5.A Mutex

A easy and naive solution is to simply wrap a container's inner hashmap in a mutex for instant, safe concurrency. However, with a mutex also comes decreased read throughput. Only one reader *or* writer may access a container's contents at a time, which falters read-heavy workloads (which are common) despite the minimal overhead incurrent by mutexes when compared to other concurrency approaches.

3.3.5.B Read-Write Lock

A slight improvement over the mutex, read-write locks allow for multiple concurrent readers and mutually exclusive writers. Read-write locks perform well in read-heavy workloads, but still suffer from one fatal limitation–a writer will block all readers, and vice versa. As such, slow writers (due to updating a possibly expansive dependency graph) *block all readers*, which is not acceptable in performant real-time applications.

3.3.5.C Transactional Map

To solve the problem of mutual exclusion between readers and writers, one can look to database management systems (DBMS). Such systems operate on data, just like containers, and often feature highly-readable transactions that operate alongside writer(s) through concurrently readable data structures. Thus, one can borrow the transaction model from DBMS to enable highly-readable workloads, while still allowing writers through multiversion concurrency control (MVCC). This solution is precisely what the Rust ReArch implementation employs; concread is the library of choice for the concurrently readable hashmap [15].

Readers are cheap and are given a consistent capsule state to read from, while writers are blocked by a mutex and incur some slight extra cost by copying data on writes to remain concurrent to any reader(s).

3.3.5.D Bucket-Based Locking

As mentioned in the last section, when using concurrently readable data structures, writers are mutually exclusive (and in the case of conread, via a mutex). This is acceptable when the capsule dependency graph may change in unforseeable ways during rebuilds (which is the case for the two accompanying ReArch implementations as capsule dependencies are allowed to arbitrarily change between builds). However, in an optimal world, one would know exactly which capsules might need to be updated during a rebuild, instead of having to assume any capsule may be updated. With this optimization, a container would only need to lock just those capsules, while allowing for other writers to update adjacent parts of the container. One such solution would be a hashmap with bucket-based locking, allowing multiple writers to operate concurrently on different buckets. However, this is not possible without compiler or some other build-time support due to the dependency on compile-time introspection to enable a reflective look at the capsule dependency graph *before* runtime.

3.3.6 Idempotent Garbage Collection

As a large application runs and references a multitude of single-purpose capsules, one may eventually want to clear the unused memory in the application's container. In addition to this growing memory footprint, bloated containers also result in increased computation; whenever a common dependency capsule rebuilds, all dependents must be rebuilt as well, even when not in use.

To mitigate these negative effects, containers (by default) perform *idempotent garbage collection* on a rebuilding capsule's dependents. Idempotent garbage collection is employed to reduce a container's memory footprint *and* skip (possibly many) capsule builds, all at *no extra cost.*

Idempotent garbage collection is built upon the idea that non-idempotent capsules are stateful and must be kept in a container's cache, preventing their disposal. Further, a non-idempotent capsule must receive all dependency updates in order to maintain *side effect correctness* (e.g., a side effect may write logs to disk; skipping log entries is unacceptable). While non-idempotent capsules *cannot be disposed* during a container's lifetime, the same cannot be said about idempotent capsules. The key observation is that idempotent capsules are entirely pure and without side effects; in other words, idempotent capsules can be procured on-demand at any time given the state of their dependencies. Thus, when a capsule rebuilds, a container is then able to identify and dispose entire idempotent subgraphs, as they can just be recreated later if/when requested. This subgraph disposal is one (common) example of idempotent garbage collection; skipping capsule builds during a rebuild, in addition to all of those capsules' future builds as well (since the subgraph will have been removed from the graph), saves many cycles of computation. This exact insight is what allows ReArch to effectively operate as an incremental computation framework; *ReArch is demand-driven*, as outlined in Adapton [11].

3.3.6.A Identifying Idempotent Subgraphs

Not all idempotent subgraphs qualify for garbage collection; the dependency relationships of all transitive dependencies of a non-idempotent capsule must be kept in the container at all times in order to preserve side effect correctness. Thus, we need an algorithm that will identify all "leaf" idempotent subgraphs that do not have dependent non-idempotent capsules. Such a task seems complicated at first glance, until one realizes that this can be solved in $O(n)$ time using a *reversed* topological ordering. As can be seen in Figure 8, one can start at the last (10th) capsule in the topological ordering and work backwards to determine which capsules only have idempotent dependents (and consequently are disposable). A Dart code example of this process is also provided in Listing 3.

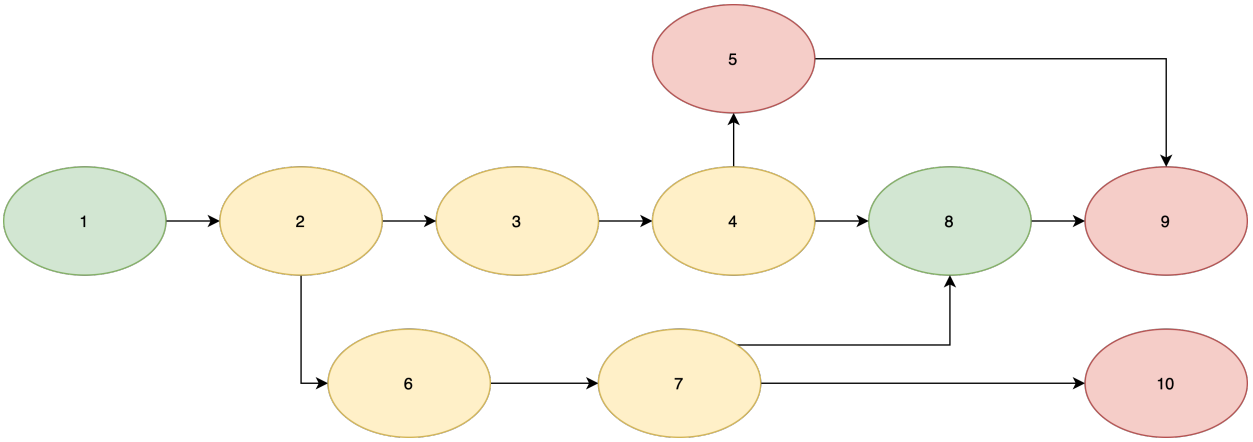

Figure 8: A set of capsules numbered in their topological ordering. Features non-idempotent capsules (green), non-disposable idempotent capsules (yellow), and disposable idempotent capsules (red).

```
Set<DataflowGraphNode> getDisposableNodesFromTopologicalOrder(
  List<DataflowGraphNode> topologicalOrder,
) {
  final disposable = <DataflowGraphNode>{};

  topologicalOrder
    .reversed
    .where((node) {
      final dependentsAllDisposable =
          node._dependents.every(disposable.contains);
      return node.isIdempotent && dependentsAllDisposable;
    })
    .forEach(disposable.add);

  return disposable;
}
```

Listing 3: A function that consumes a topological order and returns a set of its disposable idempotent capsules. Note that iterables in Dart are lazy, so `forEach` is called for every element *before* the next element is pulled and processed.

3.3.6.B Performing Idempotent Garbage Collection

Once one obtains the set of disposable nodes for a given topological ordering, the building and disposal of the appropriate capsules must be executed in order. A naive solution would be to simply build all non-disposable capsules in order and then dispose all disposable capsules. How-

ever, if a capsule that builds does not change (i.e., it returns the same data), there is no need to rebuild/dispose its dependent subgraph. To recognize this optimization, the Dart ReArch implementation rebuilds capsules in a manner similar to that shown in Listing 4.

```
void buildSelfAndDependents() {
  // We must build self, so we preemptively build it before other checks
  final selfChanged = buildSelf();
  if (!selfChanged) return;

  // Build or garbage collect (dispose) all remaining nodes
  // (We use skip(1) to avoid building this node twice)
  final topologicalOrder = createTopologicalOrder().skip(1).toList();
                           final             disposableNodes             =
getDisposableNodesFromTopologicalOrder(topologicalOrder);
  final changedNodes = {this}; // we built self above
  for (final node in topologicalOrder) {
    final haveDepsChanged = node._dependencies.any(changedNodes.contains);
    if (!haveDepsChanged) continue;

    if (disposableNodes.contains(node)) {
      node.dispose();
      changedNodes.add(node);
    } else {
      final didNodeChange = node.buildSelf();
      if (didNodeChange) {
        changedNodes.add(node);
      }
    }
  }
}
```

Listing 4: How a capsule rebuilds in Dart, featuring builds in topological order, idempotent garbage collection, and skipping builds when capsule data doesn't change.

3.3.6.C Configuring Idempotent Garbage Collection

So far, this thesis has only discussed the idempotent garbage collection that occurs during capsule rebuilds. It is possible, however, to execute idempotent garbage collection at any time. An example of this would be *eager idempotent garbage collection*, which disposes idempotent capsules directly after being read from a container (or never even persists them to the container's

cache in the first place) to reduce the container's memory footprint to the theoretical minimum. The downside of eager idempotent garbage collection, though, is that multiple reads of an idempotent capsule would cause it to be built multiple times and not cached, consuming redundant CPU cycles. Seeing as it is unlikely to ever be needed in practice (except perhaps on very memory-constrained embedded devices), eager idempotent garbage collection is currently unimplemented in both the Dart and Rust ReArch implementations.

3.4 Side Effects

Side effects, as mentioned in the capsule design section above, are a tuple of some private data and a way to mutate that data. In this model, mutations of the private data trigger a rebuild of the side effect's capsule (which in practice is done by the container providing a weak reference of itself to side effects so they may trigger the rebuild). Side effect state is stored alongside capsule data in the container, allowing the container to supply capsules with the state of their side effects whenever they are built. In other words, side effects and containers have a very close-nit relationship, as containers orchestrate the actions of side effects.

3.4.1 Example

For the most simple side effect example, take Listing 5. The container will provide `countManager` with its side effect's current state (`count`) and a way to change that count/trigger a rebuild (`setCount`). Then, `countManager` will take the output of its side effect and morph it slightly to form its own output data (`({int count, void Function() incrementCount})`). This is a fairly common pattern in ReArch; the output of a capsule can be directly provided by a side effect while providing the side effect with new semantic meaning.

```
/// A capsule that manages a counter.
({int count, void Function() incrementCount}) countManager(CapsuleHandle use) {
  final (count, setCount) = use.state(0);
  return (
    count: count,
    incrementCount: () => setCount(count + 1),
  );
}
```

Listing 5: A capsule utilizing the `state` side effect in the Dart ReArch implementation. For readers familiar with React, this is similar to `[counter, setCounter] = React.useState(0);`.

3.4.2 Self-Read Equivalence

Many side effects can also be achieved exclusively through capsule self-reads when the capsule itself is provided a rebuild mechanism, as both approaches provide the same underlying functionality. To illustrate this phenomenon, take Listing 6, which showcases two ways to count the number of changes a particular `x_capsule` undergoes.

```
fn x_capsule(_: CapsuleHandle) -> u32 {
    0
}

fn x_changes_side_effect_capsule(
    CapsuleHandle { mut get, register }: CapsuleHandle
) -> u32 {
    get(x_capsule); // mark as dep so we get updates

    let changes = register(side_effects::value(0));
    *changes += 1;
    return *changes;
}

fn x_changes_self_read_capsule(
    CapsuleHandle { mut get, register }: CapsuleHandle
) -> u32 {
    get(x_capsule); // mark as dep so we get updates

    let is_first_build = register(side_effects::is_first_build());
    if is_first_build {
        1
    } else {
        get(x_changes_self_read_capsule) + 1
    }
}
```

Listing 6: A demonstration of two different capsules that can be used to track the number of changes an `x_capsule` undergoes with the Rust ReArch implementation. The first change counter uses the `value` side effect (gives the capsule the same value across builds), whereas the second change counter uses self reads.

However, the current side effects model is more intuitive and easier to work with than self-reads directly in most cases, so side effects are provided in the accompanying ReArch implementations as the go-to solution for varying state over time.

3.4.3 Composition

Even though composing capsules forms a dependency graph, composing side effects always forms a *tree*. This important distinction allows side effects to have their own internal side effects that can be wrapped around and composed. An easier-to-understand example of this is shown in Figure 9.

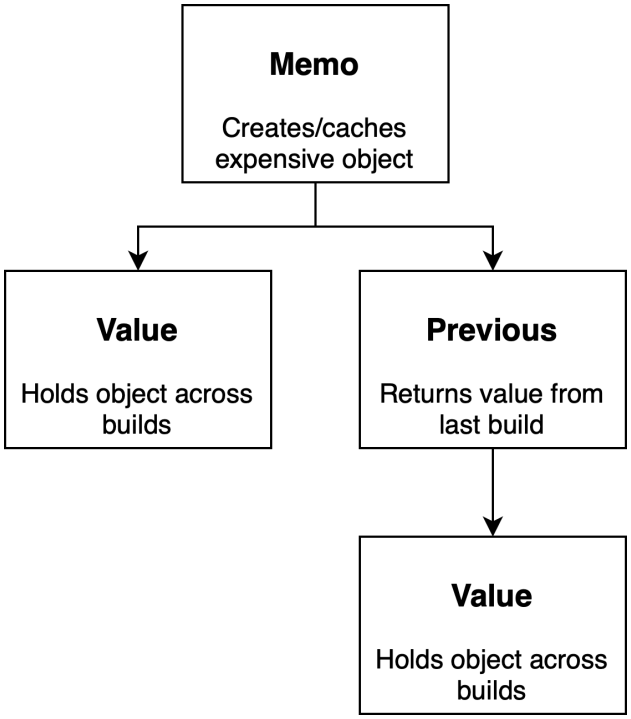

Figure 9: The memo side effect, which creates/caches some object between capsule builds, is composed from two other side effects: value and previous. Value holds the cached object itself, whereas previous is used to track dependencies.

While the memo side effect executes some logic of its own to handle object caching, it relies upon two other side effects to actually persist some state. The value side effect memo registers is used to track the cached objec itself, and the registered previous side effect is used to keep track of the cached object's dependencies. Previous in turn requires its own value side effect, to keep track of its value from the last build. Such an example demonstrates side effect composition, and this concept can be taken further to make otherwise complicated side effects with ease. In code, the memo and previous side effects can be found in Listing 7.

```
T? previous<T>(T current) {
  final (getter, setter) = use.rawValueWrapper<T?>(() => null);
  final prev = getter();
  setter(current);
  return prev;
}

T memo<T>(T Function() memo, [List<Object?> dependencies = const []]) {
  final oldDependencies = use.previous(dependencies);
  final (getData, setData) = use.rawValueWrapper<T>();
  if (_didDepsListChange(dependencies, oldDependencies)) {
    setData(memo());
  }
  return getData();
}
```

Listing 7: The source code for the memo and previous side effects in the Dart ReArch implementation. Both are formed via side effect composition, internally depending upon the `rawValueWrapper` side effect.

3.4.4 Alternatives

ReArch went through *dozens* of design overhauls and revisions before it arrived at where it is today. A couple of previous iterations of side effects were modeled as "managers" and "local capsules."

3.4.4.A Managers

Managers were originally introduced to fill the void of side effects. At one point in development, I had the idea to introduce side effects as they are today, but I dismissed the idea as "redundant," because returning some data from a capsule that is directly from a side effect seemed like an odd pattern. The alternative, however, was managers, which force the use of OOP and have a clunky/complicated interface, implementation challenges aside. Managers work by having a separate "manager" created for each capsule, where users can supply their own custom managers to work with a particular capsule. Managers can register plugins, which are composable and give the user some functionality, but with time this approach proved to be too complex and a simpler alternative was needed.

3.4.4.B Local Capsules

Before side effects were designed to be entirely separate from the lifecycle of a capsule (to enable loose coupling between the two higher level components), side effects were actually treated as

local capsules where a *global capsule* (a regular capsule in ReArch today) can have any number of global and local capsule dependencies, and a local capsule could have any number of local capsule dependencies. Local capsules would be instantiated individually on each use, which also results in a tree as seen in today's side effects.

4 Implementations

ReArch was implemented as a library for Dart (with an auxilary Flutter companion package) and Rust. With each library implementation, numerous paradigms were discovered through the building of a variety of applications.

4.1 Paradigms

Throughout the development of numerous applications, both examples and applications used in a production setting, several paradigms centered around ReArch's functional and composable nature quickly emerged.

4.1.1 UI Application State

In UI applications, especially those modeled as a tree of views, there are two overarching types of state: global and ephemeral [16]. As apps grow in complexity, there also tends to be a third: scoped state, which can have some overlap between the two other types. Scoped state is some state that is available only to the descendants (transitive closure over children) of a particular node in the view tree.

Many state management frameworks tend to overlook how to properly handle all three types, and instead focus on one or two, leaving the other(s) with half-baked solutions or ignored entirely. This is one reason that helps set ReArch's Dart/Flutter implementation apart; ReArch acknowledges that there are different types of state and that they are all applicable for different scenarios. Further, ReArch provides the same API to work with all three types, so a developer can easily transition between types when needed. An example of the three different types of state can be found in Listing 8.

```dart
// Global state (which are just capsules!)
(int, void Function()) countManager(CapsuleHandle use) {
  final (count, setCount) = use.state(0);
  return (count, () => setCount(count + 1));
}

// Ephemeral state (generates a Flutter Widget that
// has local state)
@rearchWidget
Widget localCount(WidgetHandle use) {
  final (count, setCount) = use.state(0);
  return TextButton(
    onPressed: () => setCount(count + 1),
    child: Text("I've been clicked $count times"),
  );
}

// Scoped state (generates a Flutter InheritedWidget that
// can be accessed by tree descendants)
@inheritedRearchWidget
(int, void Function()) scopedCount(WidgetHandle use) {
  final (count, setCount) = use.state(0);
  return (count, () => setCount(count + 1));
}
```

Listing 8: A snippet demonstrating how global, ephemeral, and scoped state are handled in the Dart/Flutter ReArch implementation[2].

Interestingly, with the three types of state demonstrated above, a full UI application can be made without the direct use of any object-oriented programming in application code.

4.1.2 Lexical Scoping

As capsules are just functions in their native programming language, they are accessible to only their lexical scope; i.e., a public capsule can be exposed in a library that has multiple private capsule dependencies. Because capsules are composed by referencing their dependencies, containers are able to build all private capsule dependencies without hassle, even though those capsules may not actually be in scope.

[2]The syntax shown in Listing 8 is not available at the time of writing, but should be available shortly after, dependent on when Static Metaprogramming is released.

This exact insight is what allows ReArch to work across packages/libraries, solving the distribution problem in component-based software engineering, simply by using features that already exist in modern programming languages.

For a concrete example, take Listing 9, where capsules can be made module-private. The Rust example web application follows a similar pattern to hide private capsules.

```
mod capsules {
    /// A private capsule dependency.
    fn private_dependency_1_capsule(_: CapsuleHandle) -> u8 {
        0
    }

    /// Another private capsule dependency.
    fn private_dependency_2_capsule(_: CapsuleHandle) -> u8 {
        1
    }

    /// The public capsule to be exposed outside of the module
    /// that depends upon the two private capsules.
    pub(super) fn public_capsule(
        CapsuleHandle { mut get, .. }: CapsuleHandle
    ) -> u8 {
        get(private_dependency_1_capsule) + get(private_dependency_2_capsule)
    }
}

// Bring all public capsules (only "public_capsule" here) into scope
use capsules::*;

fn main() {
  let container = Container::new();
  assert_eq!(container.read(public_capsule), 1);
}
```

Listing 9: A Rust example of how some capsules can remain private, yet still be built by a container when a public capsule is requested. Here, `public_capsule` is the only public capsule; the other two are private and are kept out of the global lexical scope.

4.1.3 Factories

Factories (also known as factory capsules) are capsules that return a function that creates some object/resource. Factories are not unique to ReArch at all, as factories are a design pattern borrowed directly from object-oriented programming.

Factories are what enables capsules to interact with non-encapsulable data, such as a text box's input. It does not make sense to encapsulate *all* data an application may encounter, especially if that data is shortlived. For these situations, factories are a perfect fit, an example of which can be seen in Listing 10.

```
String salutationCapsule(CapsuleHandle use) => 'Hello';

String Function(String) greetingFactory(CapsuleHandle use) {
  final salutation = use(salutationCapsule);
  return (name) => '$salutation, $name!';
}

final container = CapsuleContainer();
assert(container.read(greetingFactory)('Lilly') == 'Hello, Lilly!');
```

Listing 10: An example of the factory pattern in the ReArch Dart implementation. A `salutationCapsule` is used to supply the salutation used in a greeting-creation factory that is eventually invoked with an individual's name.

4.1.4 Actions

Similar to factories, actions (also known as action capsules) also return a function. However, unlike factories, actions often perform some operation that has some sort of side effect; many actions trigger container rebuilds through state updates or simply interface with some external entity (such as I/O).

The most simple action can be implemented simply to simply adapt a particular API to a capsule's consumer. Take Listing 11, where the private `_countManager` is exposed indirectly through `countCapsule`, `incrementCountAction`, and `resetCountAction`.

```dart
(int, void Function(int)) _countManager(CapsuleHandle use) => use.state(0);

int countCapsule(CapsuleHandle use) => use(_countManager).$1;

void Function() incrementCountAction(CapsuleHandle use) {
  final (count, setCount) = use(_countManager);
  return () => setCount(count + 1);
}

void Function() resetCountAction(CapsuleHandle use) {
  final (_, setCount) = use(_countManager);
  return () => setCount(0);
}
```

Listing 11: An example of the action pattern in the ReArch Dart implementation, with two actions (`incrementCountAction` and `resetCountAction`). Note that in practice, while actions can and are used like the above, often a tuple/struct (Rust) or record (Dart) can be returned from a capsule with *all* of the associated actions of a capsule to save on boilerplate. (Above, this would be a record like `({int count, void Function() incrementCount, void Function() resetCount})`.)

While Listing 11 is a valid use of actions, actions tend to be more useful when modeling individual features of an application. For example, an invoicing/billing application for a financial firm may have the following requirements:

- Load client information, including their assets
- Generate bill pdfs to send to clients
- Download/print those bills
- Print envelopes to send the bills in
- Export/display general information/statistics

All of these features are easily modeled with actions! All a developer has to do is think to make one action capsule per mental task, and an application quickly starts to fall into place.

4.1.5 Generics

As it has been mentioned numerous times, capsules are just functions that consume a `CapsuleHandle`. Due to the fact that they are just functions, capsules also follow the same rules as functions. In languages that support it, capsules can be *generic* over a set of different types.

Generic capsules are often useful when employed in conjunction with the action and factory paradigms discussed previously. A toy example of a generic capsule can be found in Listing 12.

```
fn repeated_item_factory<T: Clone>(
    _: CapsuleHandle,
) -> impl CData + Fn(T, usize) -> Vec<T> {
                  |item_to_repeat,    repetitions|    (0..repetitions).map(|_|
item_to_repeat.clone()).collect()
}

// ...
let repeated_ints_factory = container.read(repeated_item_factory::<i32>);
let repeated_strs_factory = container.read(repeated_item_factory::<&str>);
let repeated_strs = repeated_strs_factory("this will be repeated 1234 times!",
1234);
```

Listing 12: A toy example of a generic capsule in the ReArch Rust implementation that returns a list of some given item, repeated a given number of times.

While Listing 12 doesn't make sense to use outside of an example demonstrating generic capsules, the paradigm itself is very useful, especially in more advance situations. Generic capsules are used to enable Listing 17 later on, allowing database transactions to be used under a variety of circumstances.

In languages like Rust, monomorphization is used to turn each generic capsule into possibly many unique capsules. While the capsules displayed in Figure 13 later on do all technically exist only once in a mental model, a container running the application will actually hold nine "capsules":

1. Raw Database
2. With Read TXN (monomorphized for List TODOs)
3. With Read TXN (monomorphized for Get TODO)
4. With Write TXN (monomorphized for Create TODO)
5. With Write TXN (monomorphized for Delete TODO)
6. List TODOs
7. Get TODO
8. Create TODO
9. Delete TODO

While the example application only defines the With Read/Write TXN capsules once, they are automatically monomorphized when they are depended upon.[3]

4.1.6 Closures

Also referred to as *inline capsules* (as they are often defined in-line with other application code), capsules that are expressed as closures are a very useful mechanism to reduce rebuilds and/or to capture some variable(s) from the surrounding environment.

As this is a more common pattern in the Dart/Flutter ReArch implementation, the extension method shown in Listing 13 was introduced on capsules to reduce some boilerplate and possible leaks when creating inline capsules. Inline capsules are much more common when handling UI code that displays some keyed data structure.

[3] An astute reader may notice that the With TXN capsules have two generics, `F` and `R`. `F` is only ever known by the compiler, and `R` is just the return value of `F` and is known/can be explicitly written in the example application.

```dart
/// Provides the [map] convenience method on [Capsule]s.
extension CapsuleMapper<T> on Capsule<T> {
  /// Maps this [Capsule] (of type [T]) into
  /// a new idempotent [Capsule] (of type [R])
  /// by applying the given [mapper].
  ///
  /// This is similar to `.select()` in some other libraries.
  Capsule<R> map<R>(R Function(T) mapper) {
    return (CapsuleReader use) => mapper(use(this));
  }
}

// Say we have a capsule representing some list.
List<String> myListCapsule(CapsuleHandle use) => [];

// We can use the extension method above to help reduce UI rebuilds.
@rearchWidget
  Widget myListItem({
  int listIndex,
  WidgetHandle use,
}) {
  // With this inline capsule, we only rebuild when the data at
  // myList[listIndex] changes, instead of the whole myList.
  final dataAtIndex = use(
    // This creates a new inline capsule that gets a particular index of myList:
    myListCapsule.map((myList) => myList[listIndex]),

    // Alternatively, we can write the closure explicitly
    // (but this is often discouraged):
    // (CapsuleReader use) => use(myListCapsule)[listIndex],
  );
  return Text('myList[$listIndex] = $dataAtIndex');
}
```

Listing 13: An example of inline capsules in the Dart/Flutter ReArch implementation. Here, the inline capsules are used to reduce the number of (expensive) widget rebuilds.

Notice how in Listing 13 the explicitly written inline capsule is defined as `(CapsuleReader use) => ...` instead of the more brief `(use) => ...` or even `(CapsuleHandle use) => ...`. This is since `(use) => ...` would be inferred as `(CapsuleHandle use) => ...`, which itself is discouraged due

to the possibility of leaks. `CapsuleHandles` give you a `CapsuleReader` *and* `SideEffectRegistrar`, and if one is not careful, registering side effects in an inline capsule can easily lead to growing memory leaks. Instead, by manually writing `(CapsuleReader use) => ...`, we use function polymorphism to ensure leak safety by preventing the use of side effects (`CapsuleReader` is a supertype of `CapsuleHandle` in Dart).

4.1.7 Asynchrony

Often, interfacing with external systems in an application is an asynchronous operation. As an example, I/O code, such as network requests and file operations, is often expressed as asynchronous code to prevent blocking. To handle such scenarios, ReArch comes builtin with several side effects to handle common operations for asynchronous code in both the Dart and Rust implementations.

The most fundamental asynchronous side effect is for handling `Futures`; for the unfamiliar, a `Future` is an object representing the current state of a state machine modeled for some asynchronous code, where the final state is the completion of the asynchronous operation.

When used in conjunction with capsule composition, the state of asynchronous code can be easily cached within a container to be directly accessed later, as shown in Listing 14

```dart
// Some dependency capsule of the asynchronous capsule.
int countCapsule(CapsuleHandle _) => 0;

// Our async capsule that directly returns a Future.
Future<int> delayedAsyncCapsule(CapsuleHandle use) async {
  final count = use(countCapsule);

  final delayedCount = await Future.delayed(
    const Duration(seconds: 1),
    () => count,
  );

  return delayedCount + 1;
}

// We will wrap around "delayedAsyncCapsule" with this capsule that
// returns an AsyncValue, which is more useful in application code
// for handling various asynchronous states.
AsyncValue<int> delayedCapsule(CapsuleHandle use) {
  final delayed = use(delayedAsyncCapsule);
  return use.future(delayed);
}
```

Listing 14: An example of an asynchronous capsule returning a `Future`, and another capsule wrapping around the first capsule to make writing reactive, synchronous code possible based on the current asynchronous state.

Since the second capsule in Listing 14 registers a side effect (i.e., is *not idempotent*), it is eagerly updated whenever an upstream capsule is updated, ensuring the state of the lastest asynchronous computation is either computing or completed. Alternatively, without the `delayedCapsule`, `delayedAsyncCapsule` would be evaluated lazily and garbage collected, with the downside that it will not receive updates and may require time for the asynchronous computation to complete when when requested fresh from the container.

4.1.8 Warm Ups

To prevent the downtime associate with some asynchronous capsules/ to ensure an asynchronous capsule's completed state is *always* available synchronously, one can *warm up* the asynchronous capsule at application start up to cache a copy of the completed data for the remain-

der of the application. This process heavily relies upon capsule composition, as demonstrated in Listing 15.

```dart
/// The raw [SharedPreferences] async capsule to be warmed up.
Future<SharedPreferences> sharedPrefsAsyncCapsule(CapsuleHandle _) {
  return SharedPreferences.getInstance();
}

/// The warm up capsule for [sharedPrefsAsyncCapsule].
AsyncValue<SharedPreferences> sharedPrefsWarmUpCapsule(CapsuleHandle use) {
  final sharedPrefsFuture = use(sharedPrefsAsyncCapsule);
  return use.future(sharedPrefsFuture);
}

/// A synchronous copy of [sharedPrefsAsyncCapsule].
SharedPreferences sharedPrefsCapsule(CapsuleHandle use) {
  return use(sharedPrefsWarmUpCapsule).dataOrElse(
    () => throw StateError('sharedPrefsWarmUpCapsule was not warmed up!'),
  );
}
```

Listing 15: A Dart example of how to "warm up" an asynchronous capsule by wrapping around it with capsule composition. `SharedPreferences` is a common local-storage library used in Flutter applications that is only accessible asynchronously.

Similar to regular asynchronous capsules, Listing 15 starts with two capsules: one returning a `Future` and one returning an `AsyncValue`. Then, a new capsule is formed via composition that unwraps the value contained within the `AsyncValue`, throwing an error if it is not present. Thus, in order to properly read the third capsule, one must first listen to (i.e., *warm up*) the `sharedPrefsWarmUpCapsule`, and only read `sharedPrefsCapsule` once the `sharedPrefsWarmUpCapsule` emits some data (i.e., is *warmed up*). If the `sharedPrefsCapsule` is read without its dependency first being warmed up, `sharedPrefsCapsule` would raise an error during its build.

4.2 Example Applications

To showcase the process of building applications with ReArch, several open-source example applications were created. In Dart/Flutter, a to-do list application was built with a mobile-friendly UI in mind. In Rust, a to-dos web server was created as an analogous counterpart to the mobile

application and provides a REST API as its interface. Finally, the presentation for ReArch itself was built as a desktop/web application using the Dart/Flutter implementation.

4.2.1 Front End: Flutter TODOs Application

To demonstrate GUI building with the Dart/Flutter ReArch implementation, a TODOs list mobile application was made. The TODOs mobile application features a way to create, delete, search, and toggle the completion-state of a list of todos, as can be seen in Figure 10. The full source code can be found under the examples folder at https://github.com/GregoryConrad/rearch-dart.

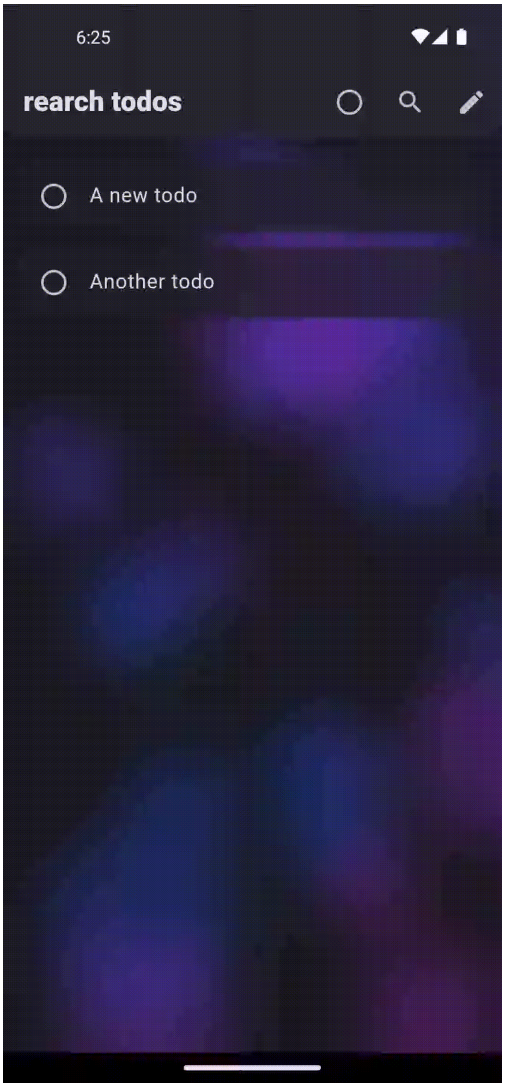

Figure 10: A screenshot of the TODOs mobile application.

To build the TODOs mobile application, a set of 7 capsules were needed as shown in Figure 11. The first three handle accessing the underlying database asynchronously (using warm up capsules as discussed previously), followed by a fourth composed capsule that provides an API to

get TODOs from the database. Then, several other capsules were made for the various UI components to operate as needed.

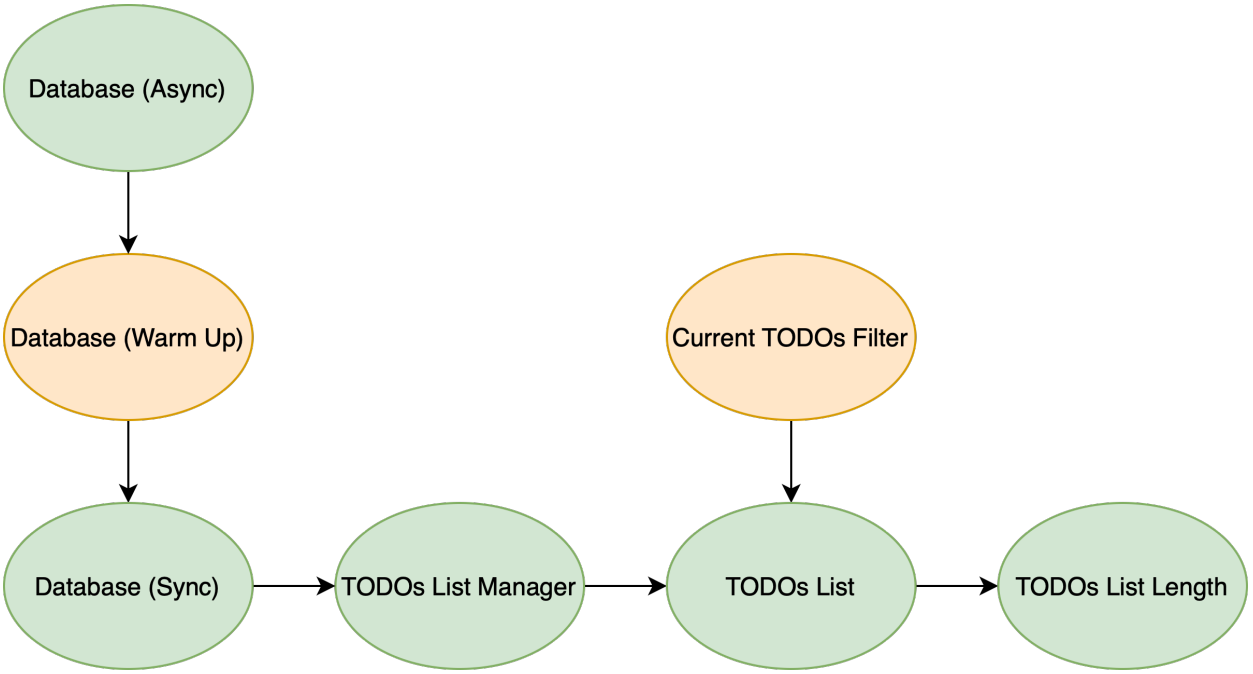

Figure 11: The underlying capsule dependency graph of the Flutter TODOs application. Capsules highlighted in yellow are non-idempotent, whereas capsules highlighted in green are idempotent.

4.2.2 Front End: ReArch Presentation

You may interact with the ReArch presentation at https://rearch-presentation.web.app and view the corresponding source code at https://github.com/GregoryConrad/rearch-dart/tree/main/examples/presentation. The left/right arrow keys move between slides and the up arrow key opens a navigation/control panel.

4.2.3 Back End: Rust TODOs Web Server

To showcase the utility of the Rust ReArch implementation, a TODOs list web server was made with Axum web framework. Axum is currently a popular choice in the async Rust ecosystem, and is a part of the tokio project [17]. The ReArch example web server serves a REST API with four available operations, as specified in Table 1. The full source code can be found under the examples folder at https://github.com/GregoryConrad/rearch-rs.

HTTP Verb	Endpoint	Description
`GET`	`/todos`	Returns list of todos
`POST`	`/todos`	Creates a new todo
`GET`	`/todos/:id`	Returns the todo with the given `id`
`DELETE`	`/todos/:id`	Deletes the todo with the given `id`

Table 1: The endpoints available on the ReArch example TODOs web server.

As such, the application can be tested by sending HTTP requests to the exposed endpoints and monitoring the state of the application.

4.2.3.A Initial Implementation

Initially, the web server was created using a total of 5 capsules. These capsules can be seen in Figure 12.

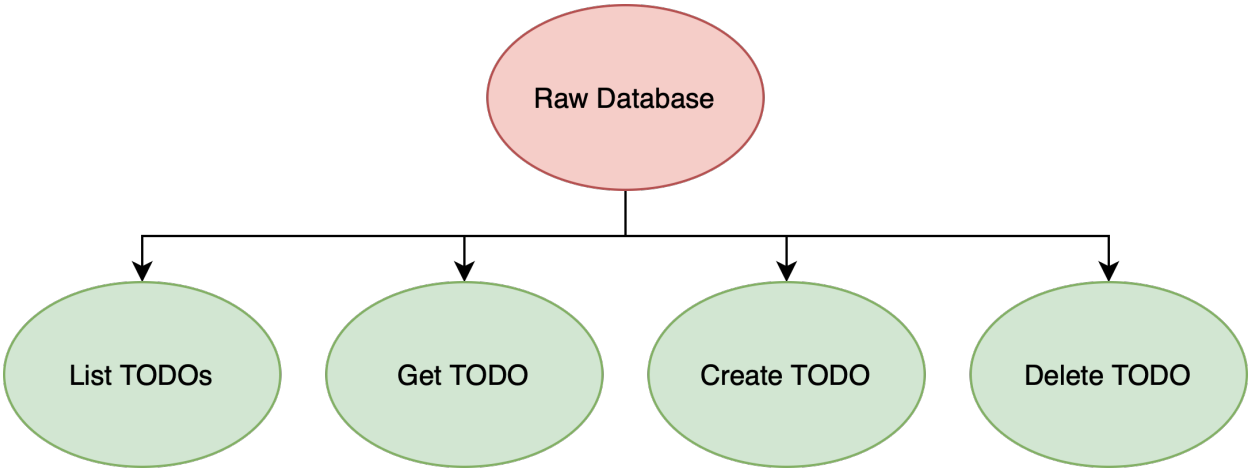

Figure 12: The initial five capsules of the ReArch example web server. Red capsules are module-private, green capsules are module-public (in terms of lexical scope).

There was one capsule wrapping around the database layer, in this case, redb, due to it's simplicity and embeddable nature [18]. This database capsule has a module-private lexical scoping, meaning that Rust code outside of the module is unable to access it. Instead, four capsules, one for each application feature, are exposed publicly in the module to be used by the web server front end. Each capsule *returns a function* that performs its designated feature; i.e., the "List TODOs" capsule returns a function that returns a list of TODOs, and the "Delete TODO" capsule returns a function that consumes an ID and deletes the corresponding TODO.

While this highly functional approach works adequately for the target needs of the TODOs server, there was a substantial amount of database boilerplate found in each feature capsule. Each feature-level capsule needed to create, mutate, and commit transactions. While this amount of

boilerplate is manageable for just four capsules, it can quickly add up and become unmaintainable as an application adds more features requiring database interoperability. For an example, see the initial "Delete TODO" capsule in Listing 16.

```
pub(super) fn delete_todo_capsule(
    CapsuleHandle { mut get, .. }: CapsuleHandle,
) -> impl CData + Fn(Uuid) -> Result<Option<String>, redb::Error> {
    let db = get.get(db_capsule);
    move |uuid| {
        let txn = db.begin_write()?;
        let mut table = txn.open_table(TODOS_TABLE)?;
          let removed_todo = table.remove(uuid.as_u128())?.map(|s|
s.value().to_owned());
        drop(table);
        txn.commit()?;
        Ok(removed_todo)
    }
}
```

Listing 16: The Rust definition of the initial "Delete TODO" capsule, demonstrating some extra database-induced boilerplate.

4.2.3.B Final Implementation

To solve the boilerplate/maintainability problems risen in the preceeding section, two intermediary capsules were added, highlighted yellow in Figure 13. These two capsules abstract away read/write transaction creation and handling logic, leaving each feature level capsule to just request/modify the data they need.

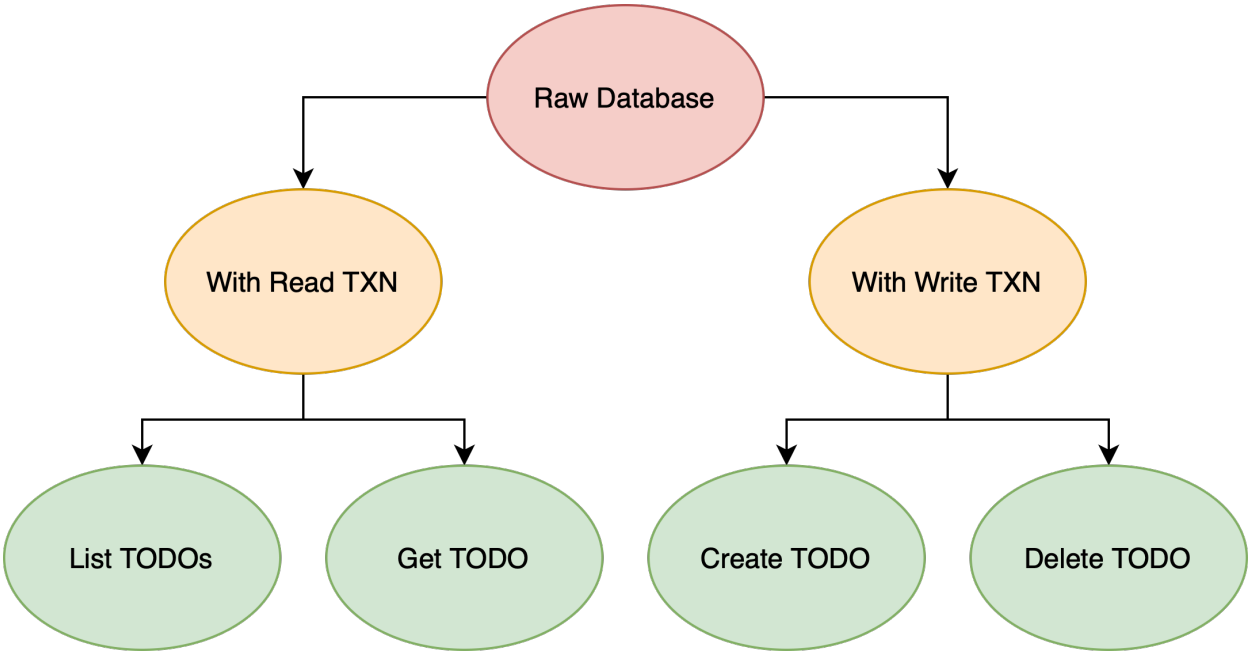

Figure 13: The final seven capsules of the ReArch example web server. Red capsules are module-private, green capsules are module-public, and yellow capsules are the additional capsules and are module-private (in terms of lexical scope).

Each capsule returns a higher order function that consumes a function to call with a given read/write transaction. The exact definitions, at the time of writing, are available in Listing 17. Each capsule consumes a `CapsuleHandle` and returns their associated higher order function (`impl Fn(F) -> Result<R, redb::Error> + Send + Sync + Clone`, where `F` is the function argument that will be invoked with a read/write transaction).

```rust
fn with_read_txn_capsule<F, R>(
    CapsuleHandle { mut get, .. }: CapsuleHandle,
) -> impl CData + Fn(F) -> Result<R, redb::Error>
where
    F: FnOnce(ReadOnlyTable<'_, u128, &str>) -> Result<R, redb::Error>,
{
    let db = get.get(db_capsule);
    move |with_table| {
        let txn = db.begin_read()?;
        let table = txn.open_table(TODOS_TABLE)?;
        with_table(table)
    }
}

fn with_write_txn_capsule<F, R>(
    CapsuleHandle { mut get, .. }: CapsuleHandle,
) -> impl CData + Fn(F) -> Result<R, redb::Error>
where
    F: FnOnce(Table<'_, '_, u128, &str>) -> Result<R, redb::Error>,
{
    let db = get.get(db_capsule);
    move |with_table| {
        let txn = db.begin_write()?;
        let table = txn.open_table(TODOS_TABLE)?;
        let result = with_table(table);
        txn.commit()?;
        result
    }
}
```

Listing 17: The Rust definitions of the “With Read TXN” and “With Write TXN” capsules, minimizing code coupling and boilerplate in the feature-level capsules.

Then, the capsules showcased in Listing 17 were used to refactor the four feature-level capsules, like the “Delete TODO” capsule, which was updated to match Listing 18.

```
pub(super) fn delete_todo_capsule(
    CapsuleHandle { mut get, .. }: CapsuleHandle,
) -> impl CData + Fn(Uuid) -> Result<Option<String>, redb::Error> {
    let with_txn = get.get(with_write_txn_capsule);
    move |uuid| {
        with_txn(move |mut table| {
            let removed_todo = table.remove(uuid.as_u128())?.map(|s|
s.value().to_owned());
            Ok(removed_todo)
        })
    }
}
```

Listing 18: The Rust definition of the final "Delete TODO" capsule, without any extra database/transaction induced boilerplate.

Such a change helps demonstrate why ReArch is applicable to many domains; it is easy to enable rapid feature adoption while keeping loose coupling simply by introducing new capsules. Further, ReArch is highly functional, and supports many functional patterns. With ReArch as an application's base, entirely purely functional applications that focus solely on data and data transformations are possible unlike before.

4.3 Rust Challenges

While the Dart ReArch implementation was relatively straightforward, the Rust implementation came with some challenges.

4.3.1 Modeling Side Effects

At a theoretical level, a side effect is merely a function that consumes a tuple of a mutable reference to it's current state and a function to rebuild a capsule while mutating its mutable state, returning an API to interact with the mutable state and/or a method to rebuild and mutate the given state. I.e., the side effect function only *transforms* its raw input into a form that is more suitable to capsules.

Naturally, a side effect should then be generic over any lifetimes for its tuple input. To properly express this in Rust using traits, one must use *Generic Associated Types* (GATs)[4]. This can look something like Listing 19.

[4]https://github.com/GregoryConrad/rearch-rs/issues/3

```rust
pub trait SideEffect {
    // Here we define the API of the side effect,
    // which should work for any lifetime 'a;
    // i.e., the side effect should not be tied to any particular tuple input.
    type Api<'a>;

    // Then, we should be able to transform the side effect tuple input
    // (wrapped in the SideEffectRegistrar below)
    // into the API above.
    fn build<'a>(self, registrar: SideEffectRegistrar<'a>) -> Self::Api<'a>;
}
```

Listing 19: Modeling side effects in Rust with a single-method trait that builds/transforms the side effect input into the side effect's API. The above example relies on GATs in order to allow the side effect to work with varied lifetimes of the `SideEffectRegistrar`.

While Listing 19 alone compiles fine, the second it is used in other code, the Rust compiler has a hard time due to limitations in the current higher-ranked lifetimes implementation. For the first example, take the `raw` side effect in <Listing 20>.

```rust
pub fn raw<T: Send + 'static>(
    initial: T,
) -> impl for<'a> SideEffect<
    Api<'a> = (
        &'a mut T,
        impl Fn(Box<dyn FnOnce(&mut T)>) + Clone + Send + Sync,
    ),
> {
    move |register: SideEffectRegistrar| register.raw(initial)
}
```

Listing 20: The proper implementation of the `raw` side effect, which simply returns the tuple of state and a way to rebuild/mutate that state as discussed previously.

While this definition looks correct, it fails to compile due to an issue with closures and lifetimes[5]. No worries, there's a workaround using nightly! One can write something along the lines of `for<'a> |b: &'a u8| -> &'a u8 { b }`. Here's the second issue: closures aren't allowed to explicity state their return type `impl Traits`, which takes this workaround out of the picture, since the second element of the tuple is an `impl Fn(Box<_>)`.

[5]https://github.com/rust-lang/rust/issues/111662

Thus, I remembered a nuanced technique I had learned previously that "helps the compiler out" with lifetimes by explicitly providing them:

```
// This will help the compiler out by providing the proper lifetime annotations.
fn fix_lifetime<F, T, R>(f: F) -> F
where
    F: for<'a> FnOnce(SideEffectRegistrar<'a>) -> (&'a mut T, R),
{
    f
}

pub fn raw<T: Send + 'static>(
    initial: T,
) -> impl for<'a> SideEffect<
    Api<'a> = (
        &'a mut T,
        impl Fn(Box<dyn FnOnce(&mut T)>) + Clone + Send + Sync,
    ),
> {
    fix_lifetime(move |register: SideEffectRegistrar| register.raw(initial))
}
```

Listing 21: A modification of the above `raw` side effect with the second workaround employed.

Now, this exact solution works for simple scenarios just fine, but for ReArch, instead fails to compile with a cryptic `error: higher-ranked lifetime error`[6].

Fast forward a couple weeks, and I have another idea: ditch the functions for now, and manually attempt to `impl SideEffect`. This would look something like Listing 22.

```
pub struct Raw<T>(T);
impl<T: Send + 'static> SideEffect for Raw<T> {
    type Api<'a> = (&'a mut T, impl CData + Fn(Box<dyn FnOnce(&mut T)>));
    fn build(self, registrar: SideEffectRegistrar<'_>) -> Self::Api<'_> {
        registrar.raw(self.0)
    }
}
```

Listing 22: A modification of the above `raw` side effect with the third (and final) workaround employed.

[6] https://github.com/rust-lang/rust/issues/116869

However, *this workaround also doesn't compile*, due to a `[E0700]: hidden type for `impl (Fn(...)) + CData` captures lifetime that does not appear in bounds`. But this doesn't make any sense; the `impl Fn` there should be `'static` (owned) as it doesn't depend on the lifetime `'a` from the higher ranked lifetime (internally, a `.clone()` is used to keep it `'static`).

All this to say: the current Rust ReArch implementation uses an incorrect model for side effects (shown in Listing 23) that ties a side effect to a particular `SideEffectRegistrar` to prevent the errors in higher ranked lifetimes by avoiding them entirely.

```
pub trait SideEffect<'a> {
  type Api;
  fn build(self, registrar: SideEffectRegistrar<'a>) -> Self::Api;
}
```

Listing 23: The current model for side effects in the Rust ReArch implementation, which is technically incorrect because the trait has a lifetime `'a` attached to it. It works, but requires an inferior/incorrect library API.

4.3.2 Differing Container Implementations

While the container provided in the Rust ReArch implementation effectively covers most needs, there are times (e.g., embedded systems/`no_std`) where a different container implementation is more appropriate. This exact scenario can be helpful in benchmarking, as seen in [19]. Unfortunately, exposing only an interface for a container, as may be done in other programming languages, would not help since differing container implementations may require different levels of thread synchronization support from capsules, which would then change the API signature. In particular, the provided container implemenation requires capsule data to be `Send + Sync + Clone + 'static`, whereas a single-threaded container may only require `Clone + 'static`. Due to this, and also reliances on container innerworkings to support side effects, the `CapsuleHandle` is not possible to port directly to other container implementations. I.e., another library could not use ReArch's `CapsuleHandle` and just provide their own container.

Instead, ReArch exposes several traits, such as `CData`, so that a user does not have to rely on manually writing `Send + Sync + Clone + 'static` and similar trait bounds in all application code. In addition to brevity, this change also enables new container implementations to simply expose the same API as ReArch, but as a polyfill so swapping the underlying implementation is still possible without affecting any application code. This poses a suitable compromise for an otherwise difficult or impossible problem.

4.3.3 Trait Upcasting

As of December 2023, the Rust ReArch implementation must be compiled with nightly due to trait upcasting[7] still requiring nightly.

Internally in the Rust implementation, a container serves as a hash map of a capsule's `TypeId` to its data. Because capsule data is stored as `dyn CapsuleData` (a trait used to represent `Any + Send + Sync + DynClone + 'static`) in the hash map to support concurrency requirements, a capsule's current data must first be upcasted to `Any` to then be downcasted to the concrete type, which is where trait upcasting comes in. Alternatively, the container implementation is possible with stable Rust and the use of unsafe Rust (pointer casting in particular), but was avoided since pointer casting, even when used carefully, can be highly dangerous.

4.3.4 Syntax Limitations

The `CapsuleHandle` is a composition of `CapsuleReader` and the `SideEffectRegistrar`, both of which simply serve as a wrapper around singular functions. Ideally, one should just be able to treat these two structs as functions, but this requires `unboxed_closures` and `fn_traits`, which will both in turn require nightly for the foreseeable future[8]. In the meantime, one must instead call `get.get(some_capsule)` and `register.register(some_side_effect())`, versus the simpler/intended `get(some_capsule)` and `register(some_side_effect())`.

4.4 Evaluations

In addition to the working example applications discussed above, the Rust ReArch implementation was also benchmarked to give a performance baseline. On my M1 Macbook Pro, the Rust ReArch implementation is able to handle >30 million container reads per second and >1.5 million container updates per second. The bottleneck, based on several flamegraphs, appears to be the underlying memory operations and allocations associated with container read and write transactions. These numbers largely correspond with best-case scenarios; under high contention, such as 8 readers a second, reads can drop to just over 2 million reads a second. Further, the 30 million reads a second is from reading a 4 byte capsule from the container; the same benchmark performed with a simple 24 byte capsule results in a significantly reduced performance of just over 20 million reads a second. Such a difference in performance helps to illustrate how important the effeciency of the underlying memory operations matter in regard to the benchmark.

[7] https://github.com/rust-lang/rust/issues/65991

[8] The two nightly features are tracked by https://github.com/rust-lang/rust/issues/29625, which is held up by the desire for variadic generics first. There is a running joke that variadic generics will be ready for use in 2030 or 2040, but I personally am excited to use variadic generics in 2050.

5 Future Works

Throughout the creation of ReArch, I had several ideas that serve as interesting follow-up ideas.

5.1 Programming Language with First-Class Capsules

Instead of defining capsules as functions, a dedicated programming language with first class capsule support would be a significant step forward for ReArch. In addition to having proper side effect support (with easier composability), such a programming language could also make numerous container-based optimizations. First, bucket-based container locking would be possible, possibly permitting some parallelism across multiple rebuilds. Further, the language could provide a generic container interface, allowing developers to easily implement their own containers for differing needs. In such a language, the container could perhaps also serve as an alternative to the heap with automatic garbage collection, which effectively solves many memory safety problems in other programming languages today.

5.2 Transactional Side Effect Mutations

ReArch currently only supports updating a singular capsule's side effect via the triggering of a rebuild. While this works for almost all real-world scenarios, it can lead to some workarounds (such as having one "central" capsule with a collection of data that other capsules then have to feed off of). With transactional side effect mutations, a developer could compose mutations across a set of capsules.

Imagine a scenario where we have two people (A and B) who each have a bank account. Say A wants to transfer some money to B. It may make sense here to model both A and B as dynamic capsules with their own funds; however, this is not possible to do transactionally with the current implementation since we cannot deduct a certain amount from A and add it to B in a single transaction. Thus, the current solution is to make one "central" capsule that houses every bank account value.

For the Dart/Flutter implementation, this should be possible to add via a new set of transactional side effects without breaking changes. However, this may require some breaking/nontrivial changes in the Rust implementation that will need to be worked out before library stabilization.

5.3 Dedicated UI Framework

A UI framework built upon ReArch's idioms would be a big leap toward enabling ReArch's adoption. Such a framework would inherently embrace functional programming without object-orientation while forcing the use of databinding. I have already drafted a prototype API for such a framework, which is visible in Listing 24.

```
fn sample_view(_: ViewHandle, _: ()) -> TerminatedView {
    view()
        // With the builtin proc macro:
        .padding(16.0) // sugar for .child(padding, 16.0) without proc macro

        // Views to align things:
        .center() // sugar for .child(center, ())

        // Support for scoping state:
        .inject(scoped_state, 1234) // injects whatever scoped_state returns into
all descendants
        // A similar macro will exist for scoping state: .scoped_state(1234)

        // And of course you can have views with multiple children:
        .row(Default::default())
        .children(vec![
            view().child(text, "Hello World!".to_owned()),
            view()
                .inject(text_style, TextStyle)
                .child(text, "Hello World Again!".to_owned()),
            view()
                .child(padding, 16.0)
                .child(column, Default::default())
                .children(vec![
                    view().child(text, "A list item:".to_owned()),
                    view()
                        .inject(scoped_index_key::<usize>, 0)
                        .child(list_item, ()),
                ]),
        ])
}
```

Listing 24: A sample view made with prototype code for a possible ReArch UI Framework. The `ViewHandle` is akin to the `CapsuleHandle`, but also contains a view-focused `Context` that allows developers to access injected/scoped state and layout constraints.

5.4 Testing Framework

As-is, ReArch remains highly testable by removing all code coupling and separating side effects. However, testing code that registers side effect(s) requires some knowledge of the inner-work-

ings of the side effect(s) in question, making testing side effects substantially more difficult. A simple testing framework that provides side effect mocks would be a nice improvement and is planned to be added to the core framework.

5.5 Eager Garbage Collection

Currently, neither library supports (a full) eager garbage collection. This decision was made on purpose, and is to enable the caching of a capsule's data to speed up future reads and more effectively serve as an incremental computation framework. However, perhaps on memory constrained devices, it may make sense to support eager garbage collection and/or to only store capsules with non-idempotent transitive dependents in the container.

6 Conclusion

ReArch is a data-driven and highly functional approach to application architecture. There are three main components: capsules, containers, and side effects. Capsules are pure functions that may register side effects in order to interface with the outside world (despite remaining functionally pure), and containers orchestrate the life cycle of capsules and any registered side effect(s), rebuilding capsules as necessary. Further, to support ReArch's functional nature, many capsules are higher order functions, treating functions and closures as data to give to other capsules, consequently enabling easy composition. As ReArch only consists of three core components, a multitude of paradigms may be employed to build more complete and complex applications that have a variety of requirements, including I/O and persistence.

To accompany the underlying theory presented in this paper, there are two supplemental ReArch library implementations, one for Rust[9], and one for Dart (with an auxiliary Flutter package)[10]. To demonstrate the utility of these libraries, there also exists several example applications, including a to-do list mobile application, a to-do list web server, and the ReArch presentation itself[11]. More information can also be found at ReArch's documentation[12].

[9] https://github.com/gregoryconrad/rearch-rs
[10] https://github.com/gregoryconrad/rearch-dart
[11] https://rearch-presentation.web.app
[12] https://rearch.gsconrad.com

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. 1994.

[2] R. F. Grove and E. Ozkan, "The MVC-web Design Pattern", in *Proceedings of the 7th International Conference on Web Information Systems and Technologies - Volume 1: WEBIST*, SciTePress, 2011, pp. 127–130. doi: 10.5220/0003296901270130.

[3] J. Kouraklis, "MVVM as Design Pattern", 2016. doi: 10.1007/978-1-4842-2214-0_1.

[4] Google, "Guice". Accessed: Dec. 08, 2023. [Online]. Available: https://github.com/google/guice

[5] "ReactiveX". Accessed: Dec. 08, 2023. [Online]. Available: https://reactivex.io/

[6] "Widget class - Dart API". Accessed: Dec. 08, 2023. [Online]. Available: https://api.flutter.dev/flutter/widgets/Widget-class.html

[7] "Salsa". Accessed: Dec. 08, 2023. [Online]. Available: https://salsa-rs.github.io/salsa/

[8] G. T. Heineman and W. T. Councill, *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley Professional, 2001.

[9] "provider | Flutter Package". Accessed: Dec. 08, 2023. [Online]. Available: https://pub.dev/packages/provider

[10] "Riverpod". Accessed: Dec. 08, 2023. [Online]. Available: https://riverpod.dev/

[11] M. A. Hammer, K. Y. Phang, M. Hicks, and J. S. Foster, "Adapton: Composable, Demand-Driven Incremental Computation", *SIGPLAN Not.*, vol. 49, no. 6, p. 156, Jun. 2014, doi: 10.1145/2666356.2594324.

[12] K.-K. Lau, M. Ornaghi, and Z. Wang, "A Software Component Model and Its Preliminary Formalisation", in *Formal Methods for Components and Objects*, F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–21.

[13] K.-K. Lau, L. Ling, and Z. Wang, "Composing Components in Design Phase using Exogenous Connectors", in *32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO'06)*, 2006, pp. 12–19. doi: 10.1109/EUROMICRO.2006.30.

[14] Facebook, "Recoil". Accessed: Dec. 09, 2023. [Online]. Available: hhttps://recoiljs.org

[15] "Concurrently Readable Data Structures for Rust". Accessed: Dec. 09, 2023. [Online]. Available: https://github.com/kanidm/concread

[16] Flutter, “Differentiate between ephemeral state and App State”. Accessed: Dec. 05, 2023. [Online]. Available: https://docs.flutter.dev/data-and-backend/state-mgmt/ephemeral-vs-app

[17] D. Pedersen, “Announcing Axum”. Accessed: Dec. 06, 2023. [Online]. Available: https://tokio.rs/blog/2021-07-announcing-axum

[18] C. Berner, “An embedded key-value database in pure Rust”. Accessed: Dec. 06, 2023. [Online]. Available: https://github.com/cberner/redb

[19] A. Krishna, A. Gokhale, D. Schmidt, D. Sevilla, and G. Thaker, “Empirically Evaluating CORBA Component Model”, 2003.