Storage Combinators

Marcel Weiher
marcel.weiher@hpi.uni-potsdam.de
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany

Robert Hirschfeld
hirschfeld@hpi.uni-potsdam.de
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany

Abstract

The ability to compose software from high level components is as sought after as it is elusive. The REST architectural style used in the World Wide Web enables such plug-compatible components in distributed settings.

We propose *storage combinators*, a type of plug-compatible component that can be used as generic intermediary in a non-distributed setting.

Storage combinators combine several stores – components that support REST-style verbs – into a single component that also provides a store interface.

This mechanism allows a few basic components to be combined in many different ways to achieve different effects with or without adaptation. It correlates with reported increases in productivity while performing well in commercial applications with millions of users.

CCS Concepts • **Information systems → RESTful web services**; • **Software and its engineering → Software architectures**; **Abstraction, modeling and modularity**; *Interoperability*; *Software performance*; *Multiparadigm languages.*

Keywords modularity, components, REST, composition

ACM Reference Format:
Marcel Weiher and Robert Hirschfeld. 2019. Storage Combinators. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '19), October 23–24, 2019, Athens, Greece.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3359591.3359729

1 Introduction

Plug-composability, where software systems, or at least parts of software systems, can be constructed by simply snapping together pre-made components like lego bricks, without time-consuming adaptation or glue code, is one of the most highly sought feature of modern software systems. It is also one of the most highly *claimed* characteristic, a claim that rarely turns out to survive closer scrutiny.

Two mechanisms that meet a high degree of syntactic composability are Unix pipes and filters and the World Wide Web (WWW). Standard Unix filters have one input interface, the `stdin` stream, and one output interface, the `stdout` stream [38][12][41]. The pipe connector is independent of any domain-specific structure or semantics. This highly reduced interface makes filters syntactically compatible by default: any filter's `stdout` can be connected to any other filter's `stdin` and the composition will be syntactically legal, though semantic compatibility is not guaranteed.

This default syntactic compatibility also makes it possible for shells to offer a composition syntax using the vertical bar that is concise, precise, and obvious: the shell program "`ls | wc`" is fully determined and specifies the relationship between the `ls` and `wc` components in this composition statically, rather than with instructions that set up the relationship.

The mechanism is closed under composition, so a composition of filters can be used like a filter in a new composition.

Similarly the REST architectural style [19] of the WWW has only a very small number of *methods*[1] (nine total), of which only two are required [18]. The small number of verbs to implement makes it easy to achieve syntactic compatibility of components. Both a simple web server and a simple web browser can be created using only the GET and HEAD methods, taking a URI as a parameter and returning header information and tagged data.

What's more, symmetric components such as proxies, caches or load-balancers can be constructed generically without having to be adapted for or even aware of any domain-specific structure or semantics. Like Unix filters, these symmetric components can be inserted between any other components without affecting further composability.

In contrast to this composability-by-default for filters and REST, the situation at least appears to be significantly more complicated in-process. Instead of unadapted composability being the default, architectural and packaging mismatch [23][17] are the norm. Many techniques are in use or have been proposed to alleviate these mismatches, for example a number of Design Patterns [22] such as Bridge and Adapater or more specialized techniques such as Binary

[1] Also referred to as *verbs*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Onward! '19, October 23–24, 2019, Athens, Greece
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6995-4/19/10...$15.00
https://doi.org/10.1145/3359591.3359729

Component Adapatation [31], though with arguably limited progress, as documented in a ten year follow-up paper to the original publication [24].

To illustrate the difference, wee will use the example of caching. In a pure HTTP setting, there are a number of ready-made web caches available such as squid [2] or varnish [5]. These caches run as separate processes. Once configured, they all have the same interfaces and can be used with any mix of unmodified HTTP clients and servers, so the choice of caching software depends on the cost/benefit analysis of the particular package, not compatibility concerns.

In the in-process case, we encounter caches in many different settings, for example HTTP servers, HTTP clients, most software that has both an in-memory and a disk representation [30], but none of these caches are interchangeable. Both the Apache [1] and nginx [3] web servers have caching modules, but they are coupled tightly to their respective servers so cannot be exchanged without changing servers, and certainly cannot be reused in clients or for non-HTTP application software.

One way to obtain composability-by-default and generic intermediaries such as caches is to architect even physically co-located systems as distributed systems in the REST architectural style, for example as *microservices* [36][21]. However, the cost of a distributed systems in time or complexity can be prohibitive, or in the case of iOS simply prohibited[2]. Looking at Fielding's evaluation of HTTP suggests that distribution may not actually be necessary [19]:

> "What makes HTTP significantly different from RPC is that the requests are directed to resources using a generic interface with standard semantics that can be interpreted by intermediaries almost as well as by the machines that originate services. The result is an application that allows for layers of transformation and indirection that are independent of the information origin, [..]"

Although the text talks about HTTP, RPC, and machines, there is nothing in the structural parts ("directed to resources using a generic interface with standard semantics") that requires a distributed implementation. Something that "[separates requests from] resources using a generic interface with standard semantics that can be interpreted by intermediaries" should make such intermediaries possible, regardless of being distributed or not.

We propose *Storage Combinators* as just such a kind of composable and (sometimes) generic intermediaries to be used and composed in-process. Based on In-Process REST[47], these intermediaries can be used to create layers of transformations similar to HTTP intermediaries in REST.

Storage combinators are specific kinds of stores, objects that implement a storage-oriented interface modelled on the REST verbs and very similar to dictionary APIs. Stores were introduced in *Polymorphic Identifiers* [48] and abstract over different kinds of storage, such as files, databases and dictionaries. Storage combinators specialise stores in that they don't just provide the storage interface to their clients, but in turn use the storage API themselves to fetch and store data.

Section 2 provides a brief introduction to the parts of In-Process REST and the Store abstraction necessary for the rest of the paper. Section 3 introduces the Store Combinator concept itself and describes the implementations of a set of combinators to be composed in the later sections. Section 4 demonstrates how complex behaviours can be obtained by plugging together combinators. Section 5 recounts practical experience using combinators and larger compositions in industry. Section 6 discusses qualitative and quantitative effects of Storage Combinator use. Section 8 shows related work and finally Section 9 gives a summary of the contribution of Storage Combinators and provides an outlook at further work.

2 In-Process REST: References, Operations, and Stores

At its most basic, taking the REST principles in-process means implementing some or all of the HTTP methods as an API in the software, and calling those methods instead of sending HTTP requests. This can be as simple as a single method that implements the equivalent of an HTTP GET by taking a string-encoded path as used by the system that introduced In-Process REST [47]. The method, shown in Figure 1, resolved the path by walking a tree of named nodes[3].

```
public SiteMapNode nodeForUri(String path) {
  return root().subobjectForPath(path);
}
...
node = nodeForUri("football/premiereLeague");
```

Figure 1. GET implementation and use in Java

The `nodeForUri()` method separates the location of the value, which is passed as a parameter to the method, from the operation, a simple fetch, which is specified in the method name. This separation mirrors the way HTTP protocol methods are defined, which also separate the verb from the URI.

This separation of location and action is necessary for Storage Combinators, but completely missing from the typical storage access mechanisms defined for programming languages, be they Strachey's *Load-Update Pairs* (LUP) defining *L-Values*, or accessors and properties in object-oriented languages, all of which define separate operations for each location to access.

[2]iOS programs are not allowed to have multiple processes

[3]Construction of the tree was handled separately.

Table 1 shows how the separate concepts, *location*, *operation*, and *resolver* are mapped to HTTP and In-Process REST, respectively. Object Oriented Progamming, on the other hand, does not separate location and operation, instead combining the two into messages and thereby limiting what an intermediary can do.

Table 1. Location, operation, and resolver

	OO	In-Process REST	HTTP
location	message	reference	URI
operation		`Storage` protocol	HTTP methods
resolver	object	store	HTTP server

In the remainder, we will use a specific implementation of In-Process REST and storage combinators to illustrate the concept. The implementation is in form of a framework [44] that supports the language Objective-Smalltalk [45].

2.1 Location: Polymorphic Identifiers/References

Polymorphic Identifiers in Objective-Smalltalk [48] are used to identify the location of a resource. They serve the same role in In-Process REST as URIs for REST/HTTP and are closely modeled on URIs.

They are also called *references* or *refs* for short. Figure 2 shows the protocol refs must conform to: a scheme and a path, which for convenience can also be accessed by components.

```
protocol Reference {
 -<Array>pathComponents.
 -<String>path.
 -<String>scheme.
}
```

Figure 2. Reference protocol in Objective-Smalltalk

Any object that conforms to the `Reference` protocol can act as a reference.

2.2 Operation: `Storage` Protocol

The equivalent of HTTP methods in REST is modelled as messages or procedure calls in In-Process REST. The implementation used here uses the messages defined in the `Storage` protocol shown in Figure 3.

```
protocol Storage {
 -at:ref.
 -<void>at:ref put:object.
 -<void>at:ref merge:object
 -<void>deleteAt:ref;
}
```

Figure 3. Storage protocol expressed in Objective-Smalltalk

The messages are equivalent to the GET, PUT, PATCH and DELETE HTTP methods, and closely mirror the protocol of Smalltalk dictionaries.

2.3 Resolver: Stores

Where HTTP servers act as the resolvers in REST, *stores* act as resolvers in In-Process REST. Any object that implements the `Storage` protocol can act as a store.

Figure 4 shows how to use a dictionary-based store via the API. First, an instance of the store is created, then the store is asked to provide a reference for a string path (references are scoped by store). Finally a data item is placed into the store using and later retrieved using messages from the `Storage` protocol.

```
store := DictStore store.
ref := store referenceForPath:'hello'.
store at:ref put:'world'.
store at:ref. // result: "world"
```

Figure 4. Using a dictionary-based store via API

Figure 5 shows the equivalent code that uses a Smalltalk dictionary directly. For this simple example, it is almost identical to the `store`-based code.

```
dict := Dictionary new
ref := 'hello'.
dict at:ref put:'world'.
dict at:ref. // result: "world"
```

Figure 5. Using a Smalltalk dictionary directly

The similarity between the dictionary-based code and the store-based code is intentional: dictionaries are often used as simple and lightweight dynamic storage mechanism. Using a dictionary-based store instead of an actual dictionary keeps the simplicity while not tying clients to the actual dictionary-based storage implementation.

Figure 6 shows how to use the a disk-store via the store API. The code is identical to Figure 4 except that `DictStore` is replaced by `DiskStore` and the effect is the same except that the data is stored in a file on disk instead of in a dictionary.

```
store := DiskStore store.
ref := store referenceForPath:'hello'.
store at:ref put:'world'.
store at:ref. // result: "world"
```

Figure 6. Using a disk-based store via API

Figure 7 demonstrates the usage of a dict-based store using Polymorphic Identifiers. Its effect is the same as the Figure 4,

but it registers the store as the handler for the scheme `dict` and then uses URIs with that scheme.

```
scheme:dict := DictStore store.
dict:hello := 'world'
dict:hello. // result: "world"
```

Figure 7. Using a dictionary-based store via Polymorphic Identifiers

Finally, Figure 8 shows the use of the disk-store via the pre-defined file scheme. Its effect is identical to Figure 6[4] .

```
file:hello := 'world'
file:hello. // result: "world"
```

Figure 8. Using a disk-based store via Polymorphic Identifiers and the pre-defined `file` scheme

Table 2 shows the four operations supported by the `Storage` protocol and their equivalent HTTP methods, Strachey LUP operations, dictionary operations, the Java In-Process-REST method shown above and equivalent Java accessors. The `at:` message is equivalent to the HTTP GET method, reading from a dictionary and fetching the specified node using `nodeForUri()` calling the `Load` function of a Load-Update Pair or calling a get accessor on a Java object.

The library comes with a number of pre-built simple stores, such as the `DictStore` and `DiskStore` already mentioned. Many more stores have already been implemented, such as those for the common URL schemes, `http`, `https`, and `ftp`, in-process schemes for environment variables (`env`) and memory variables (`var`). In addition there are stores to access databases such as key-value stores and the SQLite database, with the latter a template for connecting other relational databases. More exotic stores provide access to windows managed by the system or to other applications' data via AppleScript.

The benefits of decoupling clients from specific storage implementations via a uniform protocol roughly correspond to the general benefits of loose coupling in software engineering [50] and were discussed in detail in *Polymorphic Identifiers: Uniform Resource Access in Objective-Smalltalk* [48].

[4] Although `file:hello` is not valid URI syntax, it is a valid Polymorphic Identifier

In this specific case, the uniformity both enables the composability required for the storage combinators introduced in the next section and also incentivises it: making combinators composable and reusable is much more worthwhile if they can actually be reused in many different contexts.

3 Storage Combinators

As discussed in the previous section, any object that implements the `Storage` protocol can act as a store. This is depicted in Figure 9. There is no requirement for a store to actually store data, it can just as easily compute it, something that is common for HTTP servers, as long as it implements the protocol.

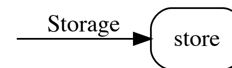

Figure 9. A store implements the storage protocol

A subset of stores compute their results by referring to other stores, combining, filtering or otherwise processing the results obtained from those other stores or being sent to those other stores. Such stores are called *storage cobinators*, shown in Figure 10.

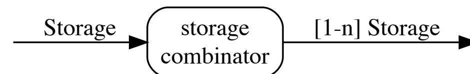

Figure 10. A storage combinator both implements and uses the storage protocol

Due to the separation of locations and meaningful operations, these storage combinators can act as intermediaries that perform useful operations, and due to their symmetry can be composed to create layers of transformation and indirection very similar to HTTP intermediaries in the REST distributed style, but without the distribution aspect.

Table 2. Operation equivalences: In-Process REST, REST, and non-REST

Store	HTTP	L-Values	Smalltalk dictionary	Java In-Proc. REST	Java accessor
store at:ref.	GET <uri>	Load	dict at:key.	nodeForUri(ref)	obj.getVar()
store at:ref put:val.	PUT <uri> <val>	Update <val>	dict at:key put:val.	-	obj.setVar(val)
store at:ref merge:val.	PATCH <uri> <val>	-	-	-	-
store deleteAt:ref.	DELETE <uri>	-	dict deleteAt:key.	-	-

3.1 Example Combinators

This section introduces a number of storage combinators that have been refined over the years and proven to be generally useful as generic intermediaries. They will also be the basis for the examples compositions in Section 4.

Figure 13 sketches a quick overview of the class hierarchy introduced here, deriving from an abstract `Store` superclass. Combinators are shown with solid outlines.

3.1.1 Pass Through Store

The simplest storage combinator is the Pass Through store. It simply passes all its requests to its source[5] store unchanged. Its implementation is shown in Figure 11. It implements every single one of the messages of the `Storage` protocol with a method that passes that message to the store's `source` and returning the result to its caller.

```
class PassThrough : Store {
   var source.
   -at:ref {
      self source at:ref.
   }
   -<void>at:ref put:object {
      self source at:ref put:object.
   }
   -<void>at:ref merge:object {
      self source at:ref merge:object.
   }
   -<void>deleteAt:ref {
      self source deleteAt:ref.
   }
}
```

Figure 11. Definition of a pass-through store

Figure 12 shows how to configure and use a pass-through store with a `DictStore` as its source. After instantiating the `PassThrough` store, we configure it by setting a `DictStore` as its source. Its use is identical to the `DictStore` example in Figure 7, and as it just passes every request through unchanged, the results are also the same.

```
scheme:p := PassThrough store.
scheme:p setSource:DictStore store.
p:hello := 'world'.
p:hello.
```

Figure 12. Configuration and use of a pass-through store

[5] We use the point of view of the GET operation to determine what is *source*.

3.1.2 Mapping Store

The Mapping Store is an abstract superclass modelled after a `map()` function [37] or a Unix filter, applying simple transformations to its inputs to yield its outputs when communicating with its source. Due to the fact that stores have a slightly richer protocol than functions or filters, the mapping store has to perform three separate mappings:

1. Map the reference before passing it to the source.
2. Map the data that is read from the source after it is read.
3. Map the data that is written to the source, before it is written.

The implementation of the mapping store is shown in Figure 14. In this particular implementation of the Mapping Store, the `Storage` protocol methods do not perform any actual mapping themselves, instead they take care of the proper sequencing of operations, deferring the mapping operation to overridable mapping methods.

The actual mapping is performed in the three methods `mapRef:`, `mapRetrieved:ref:`, and `mapToStore:ref:`. As these mapping methods are implemented as identities, returning their argument, the abstract mapper functions identically to the `PassThrough` store shown in Section 3.1.1.

The basic mapping store has stubs for these three mapping methods, which can be refined in subclasses. Having these be overridable instead of function arguments is deliberate in this particular library in order to encourage named, reusable components.

3.1.3 Relative Store

A *relative store* is a mapping store that maps references by prepending them with a prefix that is set at initialisation. Its implementation is shown in Figure 15.

Relative stores are useful for decoupling clients from absolute locations, for example a document directory or the URL of a service. The client uses relative URIs, which get mapped to the absolute URLs by the relative store. Substituting a different relative store can point the application to a test-server, or to a different directory in the filesystem. Figure 16 shows how to set up and use a disk-store

First, the code obtains a reference to the user's home directory by fetching the `HOME` environment variable and plugging that into the `file` scheme. It then creates the `home` scheme as relative store with that reference and sets that as the source of the `DiskStore` that's backing the `file` scheme. Once set up, the user's `.bashrc` file can be accessed using the expression `home:.bashrc`, assuming the user has a Unix-style operating system and a `.bashrc` file.

3.1.4 Serialiser

Where the relative store maps only references, serialisers map only data, between an external, serialised format such as JSON or XML and an internal object representation.

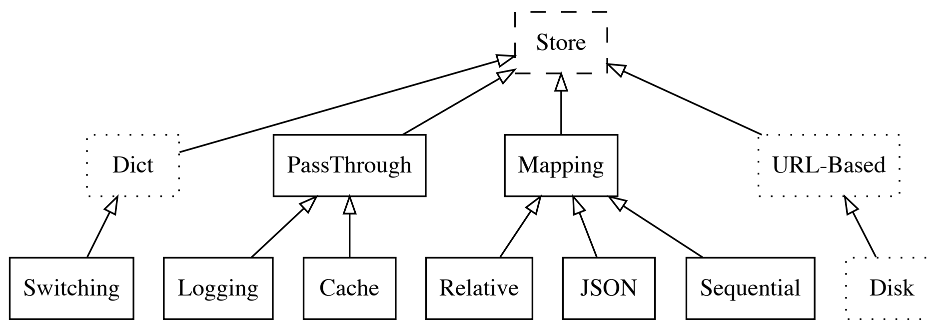

Figure 13. Stores

```
class MappingStore : Store {
   var source.
   -mapRef:ref {
        ref.
   }
   -mapRetrieved:object ref:ref {
        object.
   }
   -mapToStore:object ref:ref {
        object.
   }
   -at:ref {
       var retrieved:=(self source at: (self mapRef:
            ref )).
       self mapRetrieved: retrieved ref:ref.
   }
   -<void>at:ref put:object {
       var toStore := self mapToStore:object ref:ref:
       self source at: (self mapRef: ref ) put:toStore
            .
   }
   -<void>at:ref merge:object {
       var toMerge := self mapToStore:object ref:ref:
       self source at: (self mapRef: ref ) merge:
            toMerge.
   }
   -deleteAt:ref {
       self source deleteAt:(self mapRef:ref).
   }
}
```

Figure 14. Definition of a mapping store

3.1.5 Switching Store

The *switching store* is used to distribute requests to one of a number of subsidiary stores based on a user-defined mapping. The default mapping is to take the first path-component of the reference provided to select a store from a dictionary.

```
class RelativeStore : MappingStore {
   var prefix.
   -mapReference:ref {
       self prefix referenceByAppending: ref.
   }
}
```

Figure 15. Path relative store

```
homeRef := ref:file:{env:HOME}.
scheme:home := RelativeStore storeWithRef:homeRef.
scheme:home setSource: scheme:file.
home:.bashrc // result: <the current user's .bashrc>
```

Figure 16. A relative store pointing to the user's $HOME directory

This is comparable to a mount point with mounted file systems.

The implementation of the switching store is shown in Figure 17. It has a single method that maps the incoming ref to a store. All the methods of the `Storage` protocol than forward their request to the store returned by that method.

3.1.6 Caching Store

A caching store works as you would expect: the store has a source store and a cache store. It first tries to satisfy reads from the cache and if that fails, gets the result from its source and in addition to returning it also stores that result in the cache. On writes, it works like a write-through cache, writing to both the cache and the source. The implementation is shown in Figure 18.

It does not provide any automatic invalidation mechanism, instead adding an `invalidateReference:` message to the protocol, which removes the specified element from the cache but not from the source.

```
class SwitchingStore : DictStore {
   -storeForRef:ref {
       super at: ref firstPathComponent.
   }
   -at:ref {
      (self storeForRef:ref) at:ref.
  }
  -<void>at:ref put:object {
      (self storeForRef:ref) at:ref put:object.
  }
  -<void>at:ref merge:object {
      (self storeForRef:ref) at:ref merge:object.
  }
  -deleteAt:ref {
      (self storeForRef:ref) deleteAt:ref.
  }
}
```

Figure 17. Definition of a switching store

3.2 Logging and Filters

The *logging store* is a little different from the previously presented store combinators in that it is an adapter. Within the store combinator system, it acts as just a pass-through store, so just passes requests to its source without modification.

However, it also logs a description of the operation to its `log` instance variable. It does this by bundling the operation and the reference together into a `RESTOperation` (without the data) and then writing that object to its `log` using the `-write:` message, as shown in Figure 19. This particular logging store only logs writes.

Implementors of the `write:` message are called *filters*, and they fill roles similar to both Unix filters and output streams. As an example, the Unix `stdout` stream is mapped to a filter in Objective-Smalltalk, also called `stdout`. A similar mapping applies for `stderr`. Figure 20 shows how to configure a logging filter so it logs a textual description of write operations that reached its `DictStore`.

Beyond traditional logging, a logging store can also be used to inform other parts of a system that data has changed, and exactly which piece of data has changed. Since the information identifying the data that changed is a `ref` and not a pointer, notifications don't have to include the actual data, and the data doesn't even have to exist in memory, for example when data deleted.

3.2.1 MVC Notifications

One part of the MVC architecture [35] is a mechanism for informing the view(s) when the model has changed. The `StoreNotifications` filter transforms `RESTOperation` into MVC notifications using the `NSNotificationCenter` class of Apple's Cocoa framework.

```
class CachingStore : PassThrough {
   var cache.
   -copyFromSourceToCache:ref {
        var result := super at:ref.
        self cache at:ref put:result.
        result.
   }
   -<void>at:ref {
      var result := self cache at:ref.
      result ifNil: {
          result := self copyFromSourceToCache:ref.
      }
      result.
   }
   -<void>at:ref put:object {
      self cache at:ref put:object.
      super at:ref put:object.
   }
   -<void>at:ref merge:object {
      self copyFromSourceToCache:ref.
      var merged := self cache at:ref merge:object.
      self source at:ref put:merged.
   }
   -<void>deleteAt:ref {
      self cache deleteAt:ref.
      super deleteAt:ref.
   }
   -<void>invalidate:ref {
      self cache deleteAt:ref.
   }

}
```

Figure 18. Definition of a caching store

Any view object that has registered for this notification will be notified whenever an object in the store changes, and will also receive the reference to that object.

3.2.2 Copier

In addition to helping keep views synchronised with their model using notifications, logging stores can also be used to implement *Constraint Connectors* [49], using the `Copier` shown in Figure 22 to keep one store, the `target`, synchronised with another, the `source`.

What distinguishes constraint connectors from other one-way dataflow constraint systems is that they work with arbitrary data stores.

3.3 Discussion

This section has introduced the storage combinator concept and illustrated that concept by showing both implementations and use-cases for a number of simple and fairly generic

```
class LoggingStore : PassThrough {
   var log.
   -<void>at:ref put:object {
      super at:ref put:object.
      self log write: (RESTOperation verb:'PUT'
          reference:ref).
   }
   -<void>at:ref merge:object {
      super at:ref merge:object.
      self log write: (RESTOperation verb:'MERGE'
          reference:ref).
   }
   -<void>deleteAt:ref {
      super deleteAt:ref.
      self log write: (RESTOperation verb:'DELETE'
          reference:ref).
   }
}
```

Figure 19. Definition of a logging store

```
scheme:p := LoggingStore store.
scheme:p setSource:DictStore store.
scheme:p setLog: stderr.
p:hello := 'world'. // "PUT hello" logged to stderr
```

Figure 20. Configuring a logging store to send a textual description to `stderr`

```
class StoreNotofications {
   var notificationName.
   -<void>write:restOperation {
       NSNotificationCenter defaultCenter
                postNotificationName:self
                    notificationName
                                    object:
                                        anObject.
   }
}
```

Figure 21. A filter for sending MVC changed notifications via `NSNotificationCenter`

combinators. This list of combinators is by no means exhaustive, but it should be sufficient to clarify the ideas and serve as a basis for some compositions in the next section.

It should be noted that storage combinators cannot be viewed in isolation, they need to be combined with non-combinator stores and will frequently also be augmented by application-specific combinators. The fact that stores can unify dictionaries, file access, database access and remote interaction via HTTP give combinators very broad applicability. In Objective-Smalltalk, that applicability is extended to general variable access.

```
class Copier {
   var source.
   var target.
   -<void>write:restOperation {
        restOperation verb = 'PUT' ifTrue:{
            var ref:=restOperation reference.
            target at:ref put:(source at:ref).
        }
        restOperation verb = 'DELETE' ifTrue:{
            target deleteAt:ref.
        }
    }
}
```

Figure 22. A filter for copying between stores

4 Composition

Having defined storage combinators and shown a few very simple compositions in the previous section, this section will construct some larger compositions.

4.1 Convenience Composition Syntax

So far, composition of stores was done procedurally: a store was instantiated, configured and then connected by calling individual methods. This approach means that the static relationship between the stores is not expressed directly, instead it is hidden in the dynamic unfolding of the execution of the program, and upon reading the program has to be recovered by the reader by simulating the execution in their head.

Stores support two syntactical convenience mechanisms for constructing store compositions. The first is array-based and works in any language that supports literal arrays. The abstract store class can be initialised with an array containing either store classes or store instance. The former will first have an instance created, the latter used as is and connected in the order they appear in the array. Figure 23 constructs the same home scheme handler as Figure 16.

```
homeRef := ref:file:{env:HOME}.
scheme:home := #Store( RelativeStore storeWithRef:
    homeRef , scheme:file ).
home:.bashrc // result: <the current user's .bashrc>
```

Figure 23. Composing a $HOME-relative store via array convenience

The second syntax, only available in Objective-Smalltalk, uses right arrows to denote component connections via their default ports, and allows a few shortcuts such as using the reference directly without having to convert to a store manually, the conversion is performed as part of component adaptation that is part of the connection mechanism. Figure 24 constructs the same composite as Figure 23 and Figure 16.

```
scheme:home := ref:file:{env:HOME} -> scheme:file.
home:.bashrc // result: <the current user's .bashrc>
```

Figure 24. Composing a $HOME-relative store via connectors

4.2 HTTP Server Stack

Use in a server role is facilitated by `SchmeHTTPServer` [46], a minimal web server based on GNU libmicrohttpd [25] that maps incoming HTTP requests as trivially as possible to the `Storage` protocol given in Figure 3 (see Table 2 for the correspondences).

Given the code in Figure 24, we can minimally extend it to start serving those files in the user's home directory over HTTP on port 8080, as shown in Figure 25.

```
server := ref:file:{env:HOME} -> scheme:file ->
    SchemeHTTPServer port:8080.
server start.
```

Figure 25. A composition serving the files in $HOME via HTTP

In case serving everything from disk is too slow, we can add a cache, as shown in Figure 26. Putting just the name of a class in a composition will insert an instance of the class. The default `CachingStore` configures itself with a `DictStore` as its in-memory cache.

```
server := ref:file:{env:HOME} -> CachingStore ->
    SchemeHTTPServer port:8080.
server start.
```

Figure 26. A composition serving the files in $HOME/Sites, cached by memory

4.3 Diagrams

Compositions of stores can be represented graphically. The graphic equivalent of the composition in Figure 26 is shown in Figure 27.

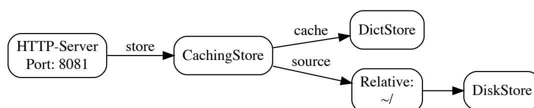

Figure 27. Diagram depicting the HTTP-server composition from Figure 26

These are architectural diagrams, with the nodes representing components and the arrows representing connectors. In this case, the rounded rectangles represent store components and the solid arrows represent connections via the `Storage` protocol.

Since the graphs have a straightforward, 1:1 correspondence to either of the convenience syntaxes presented, further compositions will be shown in their graphical representation rather than as the code constructing them.

The graphical representations shown here can be automatically generated at run time, currently by a `graphViz:` method that dumps a textual, GraphViz-compatible representation of the stores and their connections onto a stream.

A number of the graphs presented here were, in fact, generated automatically, guaranteeing their fidelity to the actual program structures. Displaying these graphs at run time has been an invaluable debugging aid. More sophisticated tools that combine the graphs with logging information (via logging store) to show live data flows at run time would be a useful extension.

4.4 HTTP Client Stack

The same cache that was used to speed up the HTTP server can also be used for the client, as shown in Figure 28.

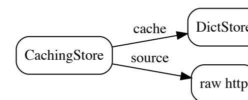

Figure 28. Composition of stores for an HTTP client with in-memory caching

A slightly more complex setup with both an in-memory and a disk-based cache is shown in Figure 29

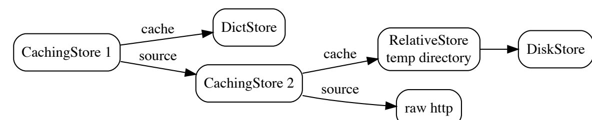

Figure 29. Composition of stores for an HTTP client with disk and in-memory caching

This composition will first try to satisfy incoming read requests from the in-memory cache, then from the disk cache and finally by requesting the original resource via HTTP.

Automatically mapping from bytes to usable objects would be accomplished by plugging in a number of serialiser stores (Section 3.1.4), with a switching store (Section 3.1.5) indexed

by MIME type selecting the appropriate serialiser. The memory cache could then be moved behind the serialisers in order to cache native objects instead of the raw bytes.

4.5 Storage Stack

One common configuration we have encountered for mobile and desktop applications is the storage stack shown in Figure 30.

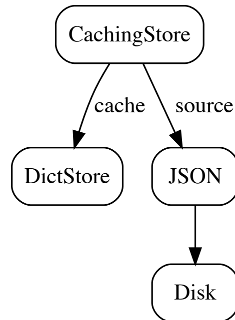

Figure 30. Common storage stack

It features an in-memory store, a disk-store with serialiser and a caching store that automatically shuffles data between disk and memory on-demand. The in-memory representation is effectively a cache of the on-disk representation.

4.6 Asynchronous Writer

One potential disadvantage of the storage stack described in the previous section is that it performs synchronous writes to disk. This is great if we want to ensure perfect consistency, but can potentially slow down operations that expect to write at memory speed.

One solution is the asynchronous writer shown in Figure 31. The ovals in this diagrams represent objects that are not stores but single-in/single-out filters discussed in Section 3.2.2.

This construction is based on the storage stack in Figure 30, but replaces the direct connection of the writing part of the `source` (disk) side of the cache with a reference to a logger (Section 3.2). For writes, instead of writing the data, the reference is to the data is entered into a queue that feeds into a a copier (Section 3.2.2). The copier copies the data from the in-memory cache where it was previously written to the its target, in this case the serializer and disk store.

The queue is asynchronous, decoupling writes to the storage stack from the speed of the underlying disk storage. The queue also coalesces writes. As the copier always gets the most current version of the data from the memory store, it does not need to write the same data multiple times.

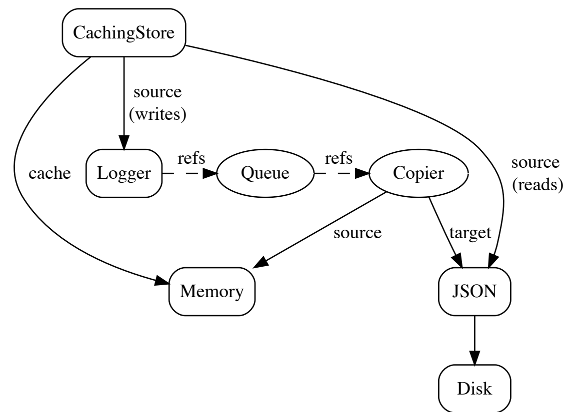

Figure 31. Asynchronous writer as a composition

The asynchronous writer was initially implemented as a single store with a very complex and error-prone procedural implementation. The realization that it could also be implemented by composing existing components only came later. The compositional implementation was both significantly simpler and more compact. It also worked out of the box after the pre-tested components were assembled.

It should be noted that the composition is hidden inside a version of the CachingStore, so for clients it has the same interfaces as CachingStore, the only difference being the asynchronous nature of writes. It should also be noted that it slightly changes the nature of the data in the cache: as long as there are writes outstanding, the data in the cache for those writes is the actual source of truth and not just a cache, and should therefore not be discarded.

A similar construction also queues up requests to web backends.

5 Experience

The stores concept was developed over several iterations of industry projects, storage combinators emerged over time as a highly useful and in-hindsight obvious addition to the mechanism.

5.1 BBC SportStats

Version 2 of the Sport Statistics system for BBC News Interactive was where the In-Process REST approach was introduced [47]. The rewrite turned a distributed system with over 100 processes and dozen machines into a single Java process. Web-sites for specific sports such as Cricket, Football or Rugby were represented by individual In-Process REST "servers" constructed out of domain objects.

A first version of the `SwitchingStore` component allowed these separate sites to be "mounted" on a single combined

"server" representing all the sports, with `/cricket`, `football` etc. "subdirectories". Each of the sites was translated to one of several output media (HTML, WAP, Ceefax) by early variants of the mapping store.

The previous system used a relational database for storage and procedural rules to map from that representation to the final output tree. Using a tree for storage removed the need for that mapping step, simplified coding and improved traceability during production, as there was a clear, static correspondence between internal representation and output.

Accessing that tree in-process rather than via an HTTP interface in a microservices architecture removed deployment complexity, removed the need for an intermediate representation that could be transformed into the final output formats and made high performance straightforward to achieve, as detailed in Section 6.

5.2 Wunderlist

Although the basic In-Process REST approach was used in several other applications, including Livescribe Mac Desktop [6], the first refinement to the actual concepts came with the the version 3 rewrite of the Wunderlist task management application for iOS and macOS [9]. The previous version 2 had used Apple's CoreData ORM (Object Relational Mapper), which had turned out to be a major source of instability, performance issues and thus ongoing rework.

There initially was no overarching plan for the storage mechanism, the team responsible for the iOS and macOS clients just implemented *The Simplest Thing that Could Possibly Work* [4] as a placeholder until we could figure out how to implement storage properly. This turned out to be a set of nested in-memory dictionaries, keyed by the type of object and the object's id.

Instead of using these dictionaries directly, we exploited the similarity between the `Storage` interface and the dictionary interface to "lift" the access to a store with minimal additional effort. By using the `Storage` interface, the team could start with a simple dictionary implementation without being tied to that implementation, for example to add actual persistence.

When the need came to actually persist data, the initial implementation grew, first to the generic storage stack from Figure 30 and then to the production version shown in Figure 32, all without having to change the interface or the client. The version shown added the ability to store some sensitive items such as credentials in the `Keychain`, a system-wide secure storage facility in macOS and iOS.

The `core` element represents the rest of the application that is the client of the storage subsystem depicted here. The `caching-store` is the same object used earlier in HTTP server and client implementations. The `switcher` is the `SwitchingStore`, keyed by object type. The object-types representing sensitive information are diverted to the keychain, the remainder written to disk as JSON.

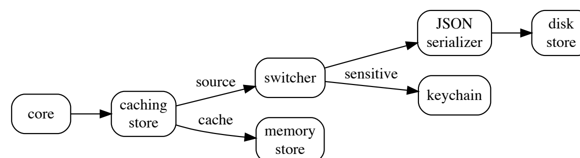

Figure 32. Composition of stores making up the Wunderlist storage architecture

Unit and system tests of application functionality use the simpler hierarchy shown in Figure 33. That way tests are fast and do not impact global state such as the developer's actual keychain.

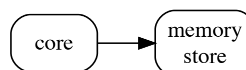

Figure 33. Simpler composition of stores making up the Wunderlist test configuration

Note that the test configuration uses no special-purpose mocks or stubs, instead creating a slightly simpler configuration of the elements that were already used elsewhere. This reduces not only the total amount of code written, but also the chances of falsifying tests due to code that runs only during test.

5.3 Microsoft To-Do

For Microsoft To-Do, started by the same team that built Wunderlist 3, the focus was expanded to domain logic and user-interface support. For example, the user interface of both Wunderlist and To-Do included a sidebar as a top-level navigational element. In Wunderlist, this was created with regular object-oriented code, in To-Do, it was generated by a mapping store.

The resulting, much more complex store hierarchy is shown in Figure 34. The `core` element once again represents the application's entry point into the store hierarchy. Actual storage management is shown near the bottom, starting with the `caching store`. It is very similar to the Wunderlist storage hierarchy shown in Figure 32, except that handling of sensitive user credentials has been delegated to Microsoft libraries and moved near the top.

The `CascadingRemoveStore` handles deleting of associated data, such as tasks in a deleted list. The `id-redirector` maps object ids assigned locally when creating an object, for example when offline, to the server-provided ids that are

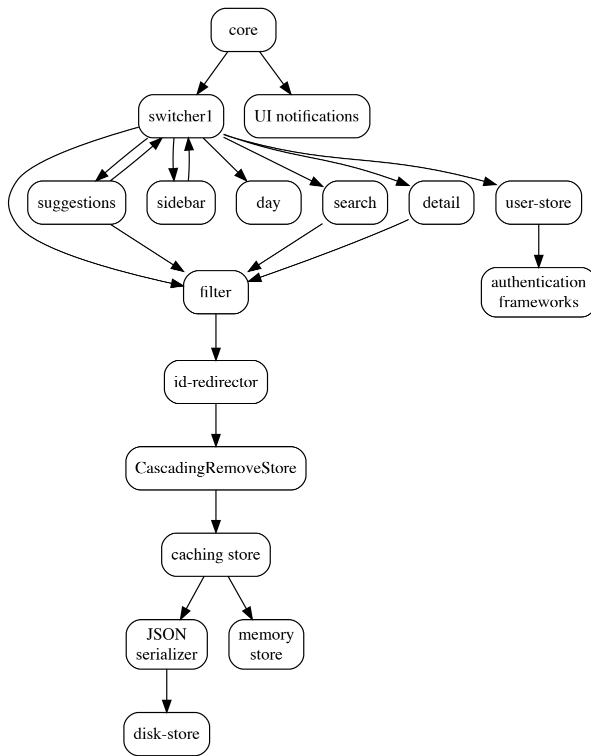

Figure 34. Composition of generic and domain-specific stores for Microsoft To-Do

substituted once the locally created object has been uploaded and assigned an id by the server.

5.4 Other

Stores and storage combinators are also in use in various projects at the Software Architecture Group at HPI, for example the Lively Kernel and Squeak Smalltalk implementations [33]. In one example, the Lively team had a performance problem with one of their stores that was fixed by plugging in a Caching Store without altering any other code [34].

The MPWFoundation project that storage combinators are a part of also contains UI components, including a miller-column-browser based on Cocoa's `NSBrowser`[6] class that works with any object conforming to the `Storage` protocol. One application is a file browser, composing the browser with the `DiskStore` using the code shown in Figure 37.

The resulting output is shown in Figure 36. Unfortunately, the combination of APIs forces quadratic complexity: the browser's data source API for fetching data is invoked separately for each individual row of each column, whereas the `DiskStore` cannot fetch individual directory entries by

[6] https://developer.apple.com/documentation/appkit/nsbrowser?language=objc

```
browser := MPWBrowser new.
fb setStore: MPWDiskStore store.
fb openInWindow:'file_browser'.
```

Figure 35. Configuring a file browser

ordinal index without reading the entire directory. For large directories, the quadratic complexity caused the UI to be severely laggy.

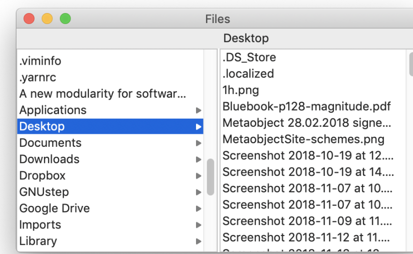

Figure 36. File Browser

The solution to the performance problems in this case was adding a caching store from Section 3.1.6 to the composition that's used to configure the browser, as shown in Figure 37. This is exactly the same cache previously used for HTTP servers, HTTP clients and for coordinating memory and disk storage in Wunderlist and To-Do. It was not adapted for this application.

```
browser := MPWBrowser new.
fb setStore: (MPWCachingStore storeWithSource:
     MPWDiskStore store cache: MPWDictStore store ).
fb openInWindow:'file_browser'.
```

Figure 37. Configuring a file browser with cache

Other applications include the server for the http://objective.st site for the Objective-Smalltalk language and several live programming environments, including one for web sites and one for iOS and macOS applications.

6 Evaluation

The REST architectural style enabled symmetric reusable components such as HTTP load balancers and caches. The goal of storage combinators was to provide similar kinds of composable intermediaries without requiring making programs distributed.

Such components can clearly be built, as shown in Section 3. Section 4 demonstrates that they can be composed.

The `CachingStore` proved to be particularly useful and versatile, being used with an HTTP server, HTTP clients in multiple configurations, in a GUI browser component, and last not least as central part of the storage layer of several widely used applications.

The straightforward composability inherent in the mechanism allowed the definition of convenience syntax that comes close to the Unix shell notation for specifying compositions in a straightforward and declarative manner.

6.1 Reliability

As pointed out by Waldo et al in their *Note on Distributed Computing*, moving from local to distributed computing introduces a number of failure modes, particularly partial failures [32]. Conversely, moving from distributed to local computing, as with storage combinators, simply removes these failure modes entirely.

The BBC Sport Statistics rewrite with storage combinators saw reliability improve by a factor of 50 as measured by production outages. While this cannot be attributed solely to the move from a distributed system with over a hundred processes spread over a dozen machines to a local system consisting of a single Java process, it certainly played a large part.

The Wunderlist 3.0 macOS/iOS release reduced crashes by a factor of ten compared to Wunderlist 2, which is unusual for an x.0 release. Crashes were reduced significantly further a couple of minor releases later after the usual x.0 oversights had been corrected. Although this was not a move from distributed to local computing, the storage subsystem was the major source of crashes in the previous version, so this improvement can be attributed at least partially to the new storage architecture, including storage combinators.

6.2 Performance

Another of the key benefits of replacing a distributed implementation with a local one is performance: the overhead of an in-process method call is orders of magnitude less than any kind Inter Process Communication (IPC).

In order to quantify this well-known difference, we compared the overhead of three HTTP-based servers (the node.js [7] based fastlify [10], the Python [42] based flask [40] and the Ruby [51] based Sinatra [15]) with storage combinators served over HTTP and used in-process.

For each, we used the minimal "hello world" program provided by the documentation, which in each case consisted of server that returns a single constant string with no further computation or marshalling overhead. The measurements were performed on a MacBook Pro 2018 with a 4 core Intel i7 CPU running at 2.7 GHz and 16GB of RAM, running Mac OS 10.14.5. For the HTTP servers, `wrk` was used to generate load.

The results are shown in Figure 38, plotted as requests per second on a log-scale.

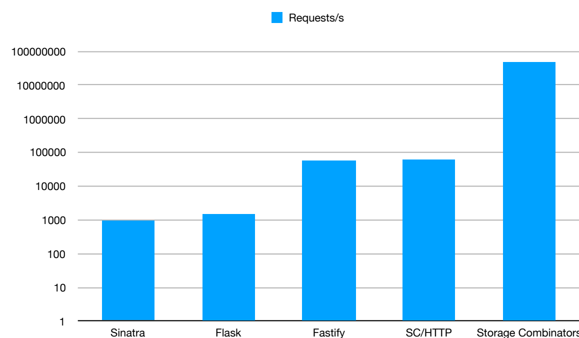

Figure 38. Overhead of Storage Combinators compared to HTTP servers (log scale)

In basic overhead, the systems compared fall into three groups, separated by 1-2 orders of magnitude in performance:

1. Sinatra and Flask with 938 and 1523 requests per second, respectively.
2. Fastlify and storage combinators over HTTP with 58120 and 63735 requests per second, respectively.
3. Storage combinators in-process with 49 million requests per second.

The in-process variant of storage combinators is 3 orders of magnitude faster than the next fastest variant, storage combinators over HTTP, and the overall range of the results is almost 5 orders of magnitude.

This comparison only accounts for the intrinsic overhead of access via the systems involved, for real systems the differences will vary depending on the actual work performed.

In production, the BBC Sports Statistics system saw a performance improvement of around a factor 100, surpassing the goal set for the rewrite and removing performance as an issue.

For the consumer-centric Wunderlist and To-Do applications we did not do a systematic, quantitative before/after analysis. We knew that performance was an issue for customers in the previous version based on qualitative customer feedback. Both ad-hoc performance analysis as well as the need to continuously tune the storage stack pinpointed the source of the problems sufficiently even without a systematic analysis.

After the rewrite, customer feedback on the topic of performance switched from complaint to rave, and the new storage stack never warranted further study in performance investigations.

6.3 Productivity

Productivity is harder to quantify than reliability and performance. One rough proxy for productivity is code size, and for the BBC Sport Statistics system, we saw a decrease in overall code size by a factor of four, despite adding two more output media (Ceefax and WAP) to the existing HTML output [47].

In Wunderlist and To-Do, both complexity and pure bulk of client code was also significantly reduced compared to the version utilising the ORM despite the ORM itself being a built-in component that does not count towards our code bulk. Behaviour became much more predictable, with team effort chasing down storage-related problems virtually eliminated.

For the more comprehensive integration of storage combinators in Microsoft To-Do, the following benefits of the architecture were reported [28]:

- "unified interface of communication between layers means the system is infinitely extensible with very little overhead of writing repeated glue code
- it gave the team a very minimal but powerful framework to communicate, work separately and successfully combine the results of their work
- specific data storage combinators could be turned on/off on the fly–.great for feature toggles and rapid development; this allowed the team to merge more often
- by applying this architecture I would be confident enough that we could rewrite Wunderlist in 50-60% of the amount of code, while introducing better testability ([..] I've managed to rewrite some of the parts in even less that 50% from the original — sidebar, tasks and detail view models)"

The downsides were reported as follows:

- "because of the unified interface and store objects being separate, it was often impossible to determine the dataflow path before runtime; dataflow would be encoded in the object dependencies, which means one has to always keep the object graph in their mind or at least before their eyes;
- often it would be difficult to debug, because a programmer has to step through the whole data transformation path during runtime"

The current Objective-C implementation of storage combinators is also fairly small, at less than 1KLOC including unit tests. Many implementations of single stores run to less than 50 LOC, again including tests.

7 Discussion

In-Process REST in general and storage combinators in particular take an architectural style known to work well in the distributed case and scale it down to work in the non-distributed, local case in order to bring along the modularity benefits associated with that style.

7.1 Inverting OO

This technique of scaling a distributed case down to the local case for structural reasons is very similar to the conception of object-oriented computing as explained by Alan Kay: "[Smalltalk's] semantics are a bit like having thousands and thousands of computers all hooked together by a very fast network." [29]. The computers are the (strongly encapsulated) objects, and they communicate by exchanging messages over the "network".

This network of thousands and thousands of computers was, at the time, mostly notional: no networks of that scale existed. Instead of such a notional network and its hypothesised communication structure, the authors were able to look at how the interconnection mechanism of a real network with billions of connected computers works in practice.

The actual architecture of the World Wide Web is based on REST, the combination of URIs to name objects and a small set of verbs with fixed, state-based semantics.

REST inverts the conventional view of object-oriented computing, which posits that in order to achieve polymorphism and encapsulation, interfaces must be procedural and "state" hidden. REST has shown that you can hide both actual state and computation behind a state-oriented interface.

In the OO view, the recursive decomposition of systems must proceed on procedural lines, because it is the procedures that make up interfaces. However, whereas procedures are a computational abstraction, storage has actually replaced computation as the main driver of data processing workloads for over a decade [13].

It is unlikely that this fundamental mismatch between what our systems do and how our tools force us to structure them is helpful for creating or understanding those systems. It could be considered an instance of *architectural mismatch*, the problems of which have been well-documented [23][24].

Allowing storage-oriented decomposition should ease this mismatch. In fact, Brooks pointed out the importance of data for system comprehension many decades ago when he wrote: "Show me your flowchart and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowchart; it'll be obvious" [16].

7.2 Architecture-Oriented Programming

Storage-oriented composition has been very successful in organising distributed systems such as the World Wide Web and, as this work shows, can also be very helpful in structuring local computation. Although one could make the case for turning this into a unifying, all-encompassing ontology similar to OO, we feel that one should refrain from making this leap.

As John Hughes wrote in *Why Functional Programming Matters* [27]:

> Modular design brings with it great productivity improvements. First of all, small

> modules can be coded quickly and easily. Second, general-purpose modules can be reused, leading to faster development of subsequent programs. Third, the modules of a program can be tested independently, helping to reduce the time spent debugging.
>
> The ways in which one can divide up the original problem depend directly on the ways in which one can glue solutions together. Therefore, to increase one's ability to modularize a problem conceptually, one must provide new kinds of glue in the programming language.

We need many different kinds of glue, many different kinds of connectors, in order to find appropriate and useful decompositions of our problems that allow us to write well-modularised programs.

8 Related Work

The Genesis project [14] was capable of synthesising a DBMS out of prefrabricated components. However, its focus was on creating a single store out of components of sub-store granularity. It did not compose subsystems with a uniform, symmetric store-like interface.

Stackable filesystems [26] are common in Unix systems nowadays. As they are services provided by the operating system kernel, they need to be provided by the kernel and either have to be compiled into the kernel itself or require special privileges to be loaded by the kernel. As such, they are typically not suitable for providing user-defined abstractions within a process. They are visible globally on a machine, so more applicable between processes, not in-process, and more typically used to structure systems, not programs. While the POSIX API to access them is also narrow and uniform, it is byte-oriented and therefore requires serialisation.

File System in User Space (FUSE) mitigates some of the problems of filesystems by providing a single in-kernel services that can be multiplexed onto multiple user-space filesystem providers. However, the general FUSE facility still requires privileges to install in the kernel, filesystems are still globally visible and require multiple processes. FUSE does allow file-oriented utilities such as `cat` and `ls` and shell scripts in general to interact directly with user-defined data structures so exposed. `MPWFoundation` provides a FUSE-adapter for exposing stores to the filesystem in this fashion.

In-Process REST shares with Plan 9 the concept of using a storage-oriented interface to represent a diverse set of storage and non-storage-oriented resources alike located in a hierarchical name space [39]. However, Plan 9 is very much an operating system[7], with the facilities provided to processes by file-servers. Unlike Unix kernel filsystems and FUSE, the name space is configured on a per-process basis. However, access does involve at least communication with a separate server process or a kernel transition when the server is provided by the kernel, and also uses a byte-oriented, rather than an object-oriented API. This makes transparent distribution transparent, but in-process use somewhat more

[7] *Plan 9 from User Space* nonwithstanding

Where recursive services analogous to storage combinators are mentioned, they are also for operating-system level services such as networking, file-serving and display/window-servers. Implementing Plan 9

HTTP middleware [43][8] provides a mechanism for filtering HTTP requests entering HTTP application servers such as then ones used in Section 6. It is similar to storage combinators in that middlewares are used in-process, can be stacked and modify requests and data, for example compressing the request body via gzip before sending. Storage combinators go beyond middleware by not being tied to a specific HTTP server, or to HTTP at all, but instead applying the REST principles to arbitrary domains.

The large number of existing middlewares almost certainly includes a rich source of inspiration for the creation or adaption of specific combinators.

The idea of moving distributed systems back into the process has also been picked up by *Fast key-value stores: An idea whose time has come and gone* [11]. The authors argue that what they call "Remote, in-memory key-value (RInK) stores" such as the popular memcached and Redis servers impose too much overhead due to marshalling costs and network hops and should be replaced by stateful application servers or custom in-memory stores. Storage combinators could help with keeping these designs modular.

Properties in languages such as C#, Objective-C, Eiffel and Swift provide a data-like interface for procedural abstraction. Unlike HTTP and in-process-REST, they do not separate the name from the operation completely, so there is no straightforward way to capture all the names for a specific operation and provide generic services such as caching or translation.

Lenses [20] share the aspect of bi-directionality with storage combinators, however they work an a per-function basis, rather than abstracting over an entire set of identifiers at once.

9 Summary and Outlook

This paper introduces *storage combinators*, generic intermediaries that can be used as part of a plug-composable storage-oriented abstraction. These composable intermediaries have proven to be compact, reusable and versatile, and are successfully used in mobile, desktop and server applications serving many millions of users. Their use correlates strongly with positive effects on code-size, performance, reliability and productivity, both observationally and in the minds of developers.

One area of future research is how to type and statically type-check storage combinators. The same generic nature that makes storage combinators so composable also makes it difficult to verify when they are connected correctly. Existing generic approaches are probably insufficient, because even single instances of the combinators will usually be processing many different types. One possibility is that the types move from the operations to the polymorphic identifiers.

Another opportunity is debugging: debugging the compositions by tracing the method invocations in their implementation can be cumbersome, but finding mechanism for directly tracing and debugging the compositions, for example by automatically inserting logging stores (see Section 3.2) could enable debugging at a higher level of abstraction than current technologies. This type of tracing could be presented graphically using a variant of the graphical representation generated.

The availability of a graphical representation that can be derived automatically from the composed stores suggests that it might be possible to create a graphical notation for *specifying* these compositions.

By combining some of the best ideas from Unix, the World Wide Web and object-oriented programming, *storage combinators* have already proven themselves to be highly versatile and effective at structuring software systems, while opening many avenues for further improvement.

References

[1] 1995. Apache Web Server. https://httpd.apache.org
[2] 1995. Squid Web Cache. http://www.squid-cache.org
[3] 2004. Nginx Web Server. https://nginx.org
[4] 2005. http://c2.com/xp/DoTheSimplestThingThatCouldPossiblyWork.html
[5] 2006. Varnish Web Cache. https://varnish-cache.org
[6] 2009. Macworld: Best of Show 2009. https://www.itworld.com/article/2782292/macworld--best-of-show-2009.html
[7] 2009. node.js. https://nodejs.org/en/
[8] 2015. HTTP Middleware. https://laravel.com/docs/5.0/middleware
[9] 2015. Wikipedia: Wunderlist. https://en.wikipedia.org/wiki/Wunderlist
[10] 2016. Fastify. https://www.fastify.io
[11] Atul Adya, Daniel Myers, Henry Qin, and Robert Grandl. 2019. Fast key-value stores: An idea whose time has come and gone. In *HotOS XVII*.
[12] M.J. Bach. 1986. *The Design of the UNIX Operating System*. Prentice-Hall. https://books.google.de/books?id=NrBQAAAAMAAJ
[13] Luiz André Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. 2000. Piranha: A Scalable Architecture Based on Single-chip Multiprocessing. *SIGARCH Computer Architecture News* 28, 2 (May 2000), 282–293. https://doi.org/10.1145/342001.339696
[14] Don Batory and Sean O'Mally. 1992. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology* (October 1992).
[15] Konstantin Haase Blake Mizerany. 2007. Sinatra. http://sinatrarb.com
[16] Frederick P. Brooks. 1975. *The Mythical Man-Month*. Addison Wesley, Reading, Mass.
[17] Robert DeLine. 1999. Avoiding Packaging Mismatch with Flexible Packaging. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*. ACM, New York, NY, USA, 97–106. https://doi.org/10.1145/302405.302456
[18] R. Fielding and J. Reschke. 2014. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. RFC Editor. http://www.rfc-editor.org/rfc/rfc7231.txt http://www.rfc-editor.org/rfc/rfc7231.txt.
[19] Roy Thomas Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. Dissertation. AAI9980887.
[20] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2005. Combinators for Bi-directional Tree Transformations: A Linguistic Approach to the View Update Problem. *SIGPLAN Not.* 40, 1 (Jan. 2005), 233–246. https://doi.org/10.1145/1047659.1040325
[21] Paolo Di Francesco, Patricia Lago, and Ivano Malavolta. 2019. Architecting with microservices: A systematic mapping study. *Journal of Systems and Software* 150 (2019), 77 – 97. https://doi.org/10.1016/j.jss.2019.01.001
[22] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
[23] David Garlan, Robert Allen, and John Ockerbloom. 1995. Architectural Mismatch: Why Reuse Is So Hard. *IEEE Softw.* 12, 6 (Nov. 1995), 17–26. https://doi.org/10.1109/52.469757
[24] David Garlan, Robert Allen, and John Ockerbloom. 2009. Architectural Mismatch: Why Reuse Is Still So Hard. *IEEE Softw.* 26, 4 (July 2009), 66–69. https://doi.org/10.1109/MS.2009.86
[25] Christian Grothoff and other authors. 2003. GNU libmichrohttpd. https://www.gnu.org/software/libmicrohttpd/
[26] John S. Heidemann and Gerald J. Popek. 1994. File-system Development with Stackable Layers. *ACM Trans. Comput. Syst.* 12, 1 (Feb. 1994), 58–89. https://doi.org/10.1145/174613.174616
[27] J. Hughes. 1989. Why Functional Programming Matters. *Comput. J.* 32, 2 (April 1989), 98–107. https://doi.org/10.1093/comjnl/32.2.98
[28] Mykhailo Karpenko. 2019. Personal Communication. (4 2019).
[29] Alan C. Kay. 1996. History of Programming languages—II. ACM, New York, NY, USA, Chapter The Early History of Smalltalk, 511–598. https://doi.org/10.1145/234286.1057828
[30] Michael Keith and Randy Stafford. 2008. Exposing the ORM Cache. *Queue* 6, 3 (May 2008), 38–47. https://doi.org/10.1145/1394127.1394141
[31] Ralph Keller and Urs Hölzle. 1998. Binary Component Adaptation. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECCOP '98)*. Springer-Verlag, Berlin, Heidelberg, 307–329. http://dl.acm.org/citation.cfm?id=646155.679694
[32] Samuel C. Kendall, Jim Waldo, Ann Wollrath, and Geoff Wyant. 1994. *A Note on Distributed Computing*. Technical Report. Mountain View, CA, USA.
[33] Marius Lichtblau, Patrick Rein, Stefan Ramson, Johannes Henning, Marcel Weiher, and Robert Hirschfeld. 2018. Squeak/Smalltalk Implementation of Polymorphic Identifiers. https://github.com/hpi-swa-lab/squeak-polymorphic-identifiers
[34] Jens Lincke. 2019. Personal Communication. (4 2019).
[35] Trygve M. H. Reenskaug. 1979. Thing-Model-View-Editor – an Example from a Planning System. http://heim.ifi.uio.no/ trygver/1979/mvc-1/1979-05-MVC.pdf. http://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pdf
[36] James Lewis Martin Fowler. 2014. Microservices: a definition of this new architectural term. https://martinfowler.com/articles/microservices.html
[37] John McCarthy. 1959. *Recursive Functions of Symbolic Expressions and Their Computation by Machine*. Technical Report. Cambridge, MA, USA.
[38] Doug McIlroy. 1964. Pipes Proposal. http://doc.cat-v.org/unix/pipes/
[39] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. 1993. The Use of Name Spaces in Plan 9. *Operating Systems Review* 27, 2 (April 1993), 72–76. http://plan9.bell-labs.com/

sys/doc/names.pdf

[40] Armin Ronacher. 2010. Flask: a Python Web Microframework. https://palletsprojects.com/p/flask/

[41] Mary Shaw and David Garlan. 1996. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[42] Guido van Rossum and Fred L. Drake. 2011. *The Python Language Reference Manual*. Network Theory Ltd.

[43] Xing Wang. 2017. What is HTTP middleware? Best practices for building, designing and using middleware. https://www.moesif.com/blog/engineering/middleware/What-Is-HTTP-Middleware/

[44] Marcel Weiher. 1998. MPWFoundation Framework. https://github.com/mpw/MPWFoundation

[45] Marcel Weiher. 2003. Objective-Smalltalk. http://objective.st

[46] Marcel Weiher. 2003. ObjectiveHTTPD Framework. https://github.com/mpw/ObjectiveHTTPD

[47] Marcel Weiher and Craig Dowie. 2014. In-Process REST at the BBC. In *REST: Advanced Research Topics and Practical Applications*, Cesare Pautasso, Erik Wilde, and Rosa Alarcon (Eds.). Springer New York, 193–209. https://doi.org/10.1007/978-1-4614-9299-3_11

[48] Marcel Weiher and Robert Hirschfeld. 2013. Polymorphic Identifiers: Uniform Resource Access in Objective-Smalltalk. In *Proceedings of the 9th Symposium on Dynamic Languages (DLS '13)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/2508168.2508169

[49] Marcel Weiher and Robert Hirschfeld. 2016. Constraints As Polymorphic Connectors. In *Proceedings of the 15th International Conference on Modularity (MODULARITY 2016)*. ACM, New York, NY, USA, 134–145. https://doi.org/10.1145/2889443.2889456

[50] Edward Yourdon and Larry L. Constantine. 1979. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design* (1st ed.). Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[51] Nobuyoshi Nakada et al. Yukihiro Matsumoto, Koichi Sasada. [n. d.]. MRI - Matz's Ruby Interpreter - The Ruby Programming Language. https://www.ruby-lang.org/en/