Can Programmers Escape the Gentle Tyranny of call/return?

Marcel Weiher
Hasso Plattner Institute, University of Potsdam, Germany

ABSTRACT

Although the call/return architectural style has served as the foundation of much of computing since its existence, it no longer matches a large proportion, probably the majority, of the programs or systems created today.

However, our programming languages, be they imperative, functional or object-oriented, support call/return variants as their primary or only abstraction mechanism. This mismatch between system structure and our means of expressing those systems can be overcome, but only with massive ("aircraft carrier") engineering effort that is beyond most casual developers.

In order to overcome this fundamental architectural mismatch and make software constructions easier for professionals and accessible for novices, we need to support other architectural styles on an equal footing with call/return in our programming languages.

This paper presents one approach to multi-architectural programming as well as progress with this approach.

CCS CONCEPTS

• **Software and its engineering → Abstraction, modeling and modularity**; **Language features**; *General programming languages.*

KEYWORDS

Means of Abstraction, Combinators, Call/Return, Constraints, Data Flow

ACM Reference Format:
Marcel Weiher. 2020. Can Programmers Escape the Gentle Tyranny of call/return?. In *Companion Proceedings of the 4th International Conference on the Art, Science, and Engineering of Programming (<Programming'20> Companion), March 23–26, 2020, Porto, Portugal.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3397537.3397546

1 INTRODUCTION

The call/return architectural style [17] has been and continues to be fundamental to virtually all mainstream programming practice and programming languages, be they imperative, functional or object-oriented. In fact, it is so foundational that its embodiment in LISP as apply/eval has been called the "Maxwells's Equations of Programming" [9], and it isn't even recognised as an underlying paradigm, with languages described as "multi-paradigm" for supporting two or more of these variants.

<Programming'20> Companion, March 23–26, 2020, Porto, Portugal

ACM ISBN 978-1-4503-7507-8/20/03…$15.00
https://doi.org/10.1145/3397537.3397546

However, the actual differences between these "paradigms" are not all that great: both imperative and functional programming have free-standing functions, sometimes called procedures that take arguments and typically return a value. In object-oriented programming, the procedures are attached to the objects, called methods and dispatched dynamically, but otherwise operate identically. Table 1 summarises the three variants:

Table 1: Comparison of "paradigms"

Paradigm	Syntax	Side effects
Imperative	f(x)	discouraged
Functional	f(x)	disallowed
Object-oriented	x.f()	discouraged/contained

However, functions, procedures and methods are primarily mechanisms for computing results, which they do, possibly given some arguments, and then terminate. However, computers and programs are no longer primarily concerned with computing results: they are used to store data and communicate, most computation occurs incidentally.

Since our computation mechanism is also our primary abstraction mechanism, we cannot abstract this difference away, leading to unresolvable mismatch.

The remainder of the paper is structured as follows: Section 2 demonstrates how call/return programming complicates even a simple temperature converter. Section 3 will look at what just happened. Section 4 presents an overview of the proposed solution and approach. Sections 5-8 describe the architectural styles that are currently supported. Section 9 shows some related work and finally Section 10 summarises.

2 MOTIVATING EXAMPLE

As an example of the architectural issues faced when assembling even fairly straightforward interactive applications, we will look at a temperature converter application that converts between different temperature scales, starting with Fahrenheit and Celsius.

2.1 Objective-Smalltalk

All code presented here is expressed in Objective-Smalltalk [18], a dialect of the Smalltalk [13] language. Objective-Smalltalk generalizes Smalltalk's support for object-oriented programming to support for defining and using architectural connectors and borrows method-definition syntax from Objective-C. Objective-Smalltalk also generalizes identifiers to Polymorphic Identifiers, which look like URIs in code [21].

Listing 1 gives a quick overview of the Objective-Smalltalk syntax: an instance method of a class is introduced with the a minus sign, followed by the method signature and the method body enclosed in curly braces. Class methods would be introduced with the plus sign, but we don't have any class methods in these examples.

```
obj msg.              // unary    obj.msg()
obj msg:7.            // keyword  obj.msg(7)
3 * 4.                // binary   3.*(4)
c := 7.               // assignment
-c {                  // instance method
   ivar:c             // return instance var c
}
ivar:field/value      // field.value;
```

Figure 1: Objective-Smalltalk Syntax

Syntax inside methods is Smalltalk, with unary, binary and keyword messages, statements terminated with periods and method return indicated by the up caret.

Identifiers look like URIs, with a scheme separated from the path by a colon, so `ivar:c` is the variable `c` in the `ivar` (instance variable) scheme. Components of a path expression are separated by the slash character typical of file systems rather than the dots more typical of languages like Java.

2.2 Basic Model

The basic model of the temperature converter consists of storage for the temperature and methods to set and inquire that temperature in different temperature scales. An implementation of the model object is shown in Listing 2. To its clients it presents an interface with two properties, `c` and `f` representing the temperature in Celsius and Fahrenheit respectively. Internally, it stores the temperature in a hidden instance variable `c` and derives the Fahrenheit value on-demand, as well as converting Fahrenheit values on input.

```
- f:degreesF {
    self c:(degreesF - 32) / 1.8.
}
- f {
    self c * 1.8 + 32.
}
- c:degreesC {
    ivar:c := degreesC.
}
- c {
    ^ivar:c.
}
```

Figure 2: Basic Temperature Converter Model

To keep the exposition manageable, we don't show boilerplate class definitions, instance variable definitions and generic application initialization code.

2.3 Connecting UI Elements

For the UI, we assume we have text fields set up to input and display numbers using Apple's Cocoa UI toolkit. Our code for hooking up those text fields to the model is shown in Listing 3. The code for creating, positioning and configuring the text fields is not shown, because we are primarily interested in the connecting code.

Lines 1-7 contain target/action methods that are called by the text fields either when the user finishes editing or on every keystroke, depending on how the fields are configured. Each target/action method has a `sender` parameter that contains a reference to the control that sent the action message. In our case, we ask the control for its numeric value and set that as the corresponding temperature, either Fahrenheit or Celsius depending on the actual control.

So far we have only had straightforward additions, but in order to update the calculated values in the UI, we have to modify our existing setter methods, both the "virtual" setter for Fahrenheit that just computes a Celsius value and the actual setter for the Celsius value.

Each of these setters updates the other UI value, setting Celsius needs to update the Fahrenheit text field, but not the Celsius text field because that is presumably where the value originated.

```
- changedF:sender {
    self f:sender intValue.
}
- changedC:sender {
    self c:sender intValue.
}
- f:degreesF {
    self c:(degreesF - 32) / 1.8.
    ivar:ui/celsiusTextField/intValue := self c.
}
- f {
     self c * 1.8 + 32.
}
- c:newValue {
    ivar:c := newValue.
    ivar:ui/fahrenheitTextField/intValue := self f.
}
- c {
    ivar:c.
}
```

Figure 3: Connecting UI via messages

While seemingly straightforward, the code in Listing 3 has a slight problem, at least if we are serious about minimizing UI updates: when setting Fahrenheit values, we call the Celsius setter after computing the correct temperature, meaning that we *do* redundantly update the UI with an already present value, at least in the case we convert Fahrenheit to Celsius.

Listing 4 fixes this problem by splitting the Celsius setter into two parts, one low-level part that doesn't do UI updates and is called when converting from Fahrenheit, and one high-level part that is called when actually entering Celsius for conversion and does do the UI update. (Only the methods that were changed are shown).

2.4 Adding Persistence

Adding persistence also requires making modifications to existing code, though only to a single method which is why Listing 5 only shows that single modified method. Whenever we set a new Celsius value, we write this value to the user defaults database.

As we can see, the cover setter method for the Celsius variable is getting to be a hub of changes for any additional architectural dependencies.

```
- f:degreesF {
    self basicC:(degreesF - 32) / 1.8.
    ivar:ui/celsiusTextField/intValue := self c.
}
- c:newValue {
    self basicC:newValue.
    ivar:ui/fahrenheitTextField/intValue := self f.
}
- basicC:newValue {
    ivar:c := newValue.
}
```

Figure 4: Minimizing UI updates

```
- basicC:newValue {
    ivar:c := newValue.
    NSUserDefaults standardUserDefaults
                   setObject:newValue forKey:'c'.
    self updateUI.
}
```

Figure 5: Persistence using native API

To make the code more comparable to the constraint solution we build in Section 5, Listing 6 shows the same code expressed with a Polymorphic Identifier using the `defaults` scheme instead of a message-send to the `NSUserDefaults` shared instance. The two pieces of code have the same semantic.

```
- basicC:newValue {
    ivar:c := newValue.
    defaults:c := ivar:c.
    self updateUI.
}
```

Figure 6: Persistence using Polymorphic Identifier

This is a very simple persistence solution, as we are only concerned about a single value, without any relationships, object identity or complex queries to worry about. In a more complex application, dealing with persistence is likely to be much more troublesome.

2.5 Adding a Temperature Scale

Adding a new temperature scale, in this case the Kelvin scale that starts at absolute zero, involves adding the conversion methods for the new scale, adding UI elements (not shown) and hooking them up to the model, shown in Listing 7.

Most of the code consists of straightforward if tedious additions, except for the code keeping the UI in sync with the model. Every method that sets a new value for a specific temperature has to be modified to update the respective other temperature text fields (assuming that the change originated in the UI).

At this point, it becomes clear that our original strategy of updating the UI from the individual setter methods is probably not

```
- changedF:sender {
    self f:sender intValue.
}
- changedC:sender {
    self c:sender intValue.
}
- changedK:sender {
    self k:sender intValue.
}
- f:degreesF {
    self basicC:(degreesF - 32) / 1.8.
    ivar:ui/celsiusTextField/intValue := self c.
    ivar:ui/kelvinTextField/intValue := self k.
}
- f {
     self c * 1.8 + 32.
}
- k:degreesK {
    self basicC:(degreesF - 273.15.
    ivar:ui/celsiusTextField/intValue := self c.
    ivar:ui/fahrenheitTextField/intValue := self f.
}
- k {
     self c + 273.15.
}
- c:newValue {
    self basicC:newValue.
    ivar:ui/fahrenheitTextField/intValue := self f.
    ivar:ui/kelvinTextField/intValue := self k.
}
- c {
    ivar:c.
}
- basicC:newValue {
    defaults:c := ivar:c.
    ivar:c := newValue.
}
```

Figure 7: Adding Kelvin scale

tenable in the long run. Listing 8 replaces this distributed logic with a centralized `-updateUI` method that updates all of the UI from the model. This method is only invoked from the `-c:` accessor method. While this change simplifies the code, it removes the optimization that prevented updating the UI element that initiated the change.

The solution in Listing 8 starts to approximate a true Model View Controller (MVC) [14][15] approach. However, the UI elements are widgets, not classical Views, so they contain their own data rather than referring to and refreshing themselves from the model. Updated data must therefore be pushed to them, they cannot pull it after receiving a `#changed` notification. Actually implementing the MVC pattern in this instance would therefore entail introducing an intermediate layer that mediates between the widgets and the model, listening to `#changed` notifications and pulling data from the model and pushing to the widgets.

2.6 Discussion

As we have seen, even a conceptually very simple application such as a temperature converter quickly attracts significant complexity

```
-updateUI {
    ivar:ui/celsiusTextField/intValue := self c.
    ivar:ui/fahrenheitTextField/intValue := self f.
    ivar:ui/kelvinTextField/intValue := self k.
}
- f:degreesF {
    self c:(degreesF - 32) / 1.8.
}
- f {
     self c * 1.8 + 32.
}
- k:degreesK {
    self c:(degreesF - 273.15.
}
- k {
     self c + 273.15.
}
- c:newValue {
    ivar:c := newValue.
    self updateUI.
}
- c {
    ivar:c.
}
```

Figure 8: Centralized UI update

with non-obvious trade-offs once the requirements of an interactive version of that application are taken into account.

This complexity is not the result of essential complexity in the domain model, but rather of the architectural embellishments required to move data from location to location in order to keep the different parts of the application (model, user interface, persistence) synchronised using methods that perform an action and then terminate.

As Guy Steele noted in the OOPSLA Panel *Resolved: Objects Have Failed* [1] (representing the position that object have not failed):

> Another weakness of procedural and functional programming is that their viewpoint assumes a process by which "inputs" are transformed into "outputs";
>
> [...] the procedural and functional models have failed, another reason why objects have become the dominant exmodel. Ongoing behavior, not completion, is now of primary interest.

The relationship between the UI and the model is an ongoing one, as is the one between Celsius and Fahrenheit and between the in-memory and persisted version. Expressing these relationships as a set of actions that need to be called to maintain those relationships adds accidental complexity that completely overwhelms the simplicity of the relationships.

3 ARCHITECTURAL MISMATCH

The problems illustrated in the previous section are not an isolated example. The fundamental mismatch between UI programming and call/return programming languages was described by Chatty [6].

However, UIs are not the only problem. Other forms of *Architectural Mismatch* are described in *Architectural Mismatch: Why Reuse Is So Hard* [12], one of these being that (procedure) imports serve multiple purposes [17]:

> ...the use of imports and exports confuses algorithmic with architectural description. When facilities are imported from another module, this may may indicate interaction between components. But it might also simply represent the inclusion of lower-level facilities to aid in the implementation of the importing module – for example, by importing a library module.

This realisation that subroutines serve multiple purposes is not new, in 1952, Wheeler noted the two distinct uses of subroutines [25]:

> Sub-routines seem to have two distinct uses in programmes. The first and most obvious use is for the evaluation of functions, a simple example being the evaluation of sine x given x. The second use is for the organization of processes [...]

As programs have shifted away from computing values, the use of a function evaluation mechanism as a program structuring mechanism has become more and more strained. In addition to the ongoing behaviour that is more typical of programs today, problems also manifest themselves in error handling or asynchronous programming where return values may not be available yet or at all.

This mismatch presents the software practitioner with a stark choice: either use the architectural style appropriate for the software under construction, and mismatched to the language, or use an architectural style matched to the language, but mismatched to the software. Neither choice is good.

4 GENERALISING TO ARCHITECTURE

The hypothesis of this work is that the way to overcome the limitations of the call/return style is neither to extend it, nor to replace it, but rather to generalise from this one particular architectural style to a method of programming that allows multiple architectural styles.

This approach reconciles the conflicting observations that call/return is, as shown above, obviously limited, but at the same time also incredibly useful, flexible and ubiquitous. It also has precedent in the natural sciences, where newer theories such a general relativity or quantum mechanics often include their classical predecessors as special cases that are still widely useful.

As programming languages tend to at most allow extension, not generalisation, testing this hypothesis requires a new (kind of) programming language, one where "multi-paradigm" is not limited to OO, FP and imperative styles. Objective-Smalltalk [18] was developed for this purpose and serves as a test-bed.

Another necessary ingredient is a set of architectural styles to adapt and incorporate. A pragmatic selection includes the aforementioned call/return, dataflow constraints, Unix Pipes and Filters and the REST architectural style underpinning the World Wide

Web. This list is not meant to be definitive, but presents a good starting point.

Finally, these architectural styles must be used in actual systems, because problems with the approach typically only manifest themselves when used at scale. This presents a conundrum as it is difficult to adopt experimental programming languages in real-wold projects. So instead, libraries or frameworks implementing the architectural styles are created with mainstream programming languages but with linguistic integration in mind, and then adapted to Objective-Smalltalk.

5 CONSTRAINT CONNECTORS

In order to illustrate the potential benefits of non-call/return architectural styles, let's implement the temperature converter example using Constraint Connectors [22], which provide a kind of one-way and two-way dataflow constraints.

The one-way constraints are expressed syntactically using the connector `|=`, which specifies that that the left hand side should be kept in sync with the right hand side. The bi-directional connector `=|=` specifies that both sides are to be kept in sync with each other.

The initial model shown in Listing 9 has mostly superficial differences from the initial example in Listing 2: instead of a single variable for Celsius, both Fahrenheit and Celsius are represented as instance variables. The constraints specifying their relationships are encoded directly and independently from setters and getters, which are hidden.

```
ivar:f |= (9.0/5.0) * ivar:c + 32 .
ivar:c |= (ivar:f - 32) * (5.0/9.0).
```

Figure 9: Basic Temperature Converter Model

The update logic is triggered automatically whenever a variable is modified to keep the other variable in sync.

5.1 Adding User Interface

The architectural differences become more noticeable when adding UI, as shown in Listing 10. The code to hook up the UI is purely additive, which is why Listing 10 only shows the additions. It defines two additional bi-directional dataflow constraints that keep the instance variables synchronized with their respective UI text fields.

```
ivar:ui/celsiusTextField/intValue    =|= ivar:c.
ivar:ui/fahrenheitTextField/intValue =|= ivar:f.
```

Figure 10: Adding UI

The original model code does not need to be updated, because the update logic is implicit in the constraint definitions. Optimally updating only the values that have changed, so for example only the dependent values when changing is also implicit in the connector definition, and implemented behind the scenes in the constraint solver.

5.2 Adding Persistence

Adding persistence is as easy as adding a constraint from one of the temperature instance variables to the persistent variable, it doesn't really matter which.

```
ivar:c := defaults:celsius.
defaults:celsius |= ivar:c.
```

Figure 11: Adding Persistence

Referring to the persistent variable using the Polymorphic Identifier `defaults:celsius` makes it possible to place it on the left hand side of a constraint connector (`|=`), something that would have been much more difficult with the method API (Listsing 5, lines 3-4).

5.3 Adding a Temperature Scale

This time, adding the Kelvin temperature scale is also purely additive. We add an instance variable `ivar:k` to hold the temperature in degrees Kelvin (not shown), connect an additional text field to that field and define additional constraints relating the Kelvin and Celsius instance variables. These additions are shown in Listing 12.

```
ivar:ui/kelvinTextField/intValue     =|= ivar:k.
ivar:k |= ivar:c + 273.15.
ivar:c |= ivar:k - 273.15.
```

Figure 12: Adding a Temperature Scale

This brings us to the (almost) complete temperature converter application shown in Listing 13. It shows all the elements of the application and how they interact. It is not only more compact than the non-constraint version in Listing 8, but actually handles a few details that were elided in that version, such as initialization from persistence and hooking up the text fields to the model.

```
ivar:ui/celsiusTextField/intValue    =|= ivar:c.
ivar:ui/fahrenheitTextField/intValue =|= ivar:f.
ivar:ui/kelvinTextField/intValue     =|= ivar:k.

ivar:f |= (9.0/5.0) * ivar:c + 32 .
ivar:c |= (ivar:f - 32) * (5.0/9.0).
ivar:k |= ivar:c + 293.
ivar:c |= ivar:k - 293.

ivar:c := defaults:celsius.
defaults:celsius |= ivar:c.
```

Figure 13: Complete Temperature Converter

The complete program is the simple concatenation of the individual pieces, no modifications of previous code was necessary.

5.4 Discussion

The point of the previous example is not that dataflow constraints are the solution, though they are almost certainly *part* of a solution. The point is that describing systems with ongoing behaviour is much more straightforward when we have linguistic means for *directly* expressing that ongoing behaviour.

But doesn't object-oriented programming provide those mechanisms, as Guy Steele posited? Not quite, as Andrew Black notes in *Object-oriented programming: Some history, and challenges for the next fifty years* [4]:

> The program's text is a meta-description of the program behavior, and it is not always easy to infer the behavior from the meta-description. [..]
>
> In my own practice as a teacher of object-oriented programming, I know that I have succeeded when students anthropomorphize their objects, that is, when they turn to their partners and start to speak of one object asking another object to do something. I have found that his happens more often and more quickly when I teach with Smalltalk than when I teach with Java: Smalltalk programmers tend to talk about objects, while Java programmers tend to talk about classes.

So although object-oriented programming allows us to express these continuous relationships, it does so only at the cost of the actual program text, which is still call/return oriented, only being a meta-description whose actual behaviour needs to be inferred, with difficulty.

To illustrate this problem, let's look at the code for setting up an equivalent binding in Apple's Cocoa framework [3] in Listing ??.

```
[celsiusField bind:@"value" toObject:model withKeyPath:@"
    ↪ celsius" options:nil];
```

Figure 14: Setting up a Cooca Binding

In addition to being stringly typed and not entirely obvious as to what is actually happening, this code *sets up* the binding, it *is not* the binding. This means that the binding happens as a side effect of sending this message, the code is the meta-desciption. There is no place in the code that is an actual description of the binding. Apart from the problems with conceptualising and visualising what the program does, it also makes debugging very difficult, because there is no line in the user code that the debugger can display when there is a problem with the binding.

And of course, having to infer dynamic program behaviour from a textual description that is far removed from said dynamic behavior is not a new problem, it was famously described by Dijkstra in *Go To Statement Considered Harmful* [8]:

> My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect;it is this process that in its dynamic behavior has to satisfy the desired specifications.Yet,once the program has been made, the "making" of the corresponding process is delegated to the machine.
>
> My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the cor- respondence between the program (spread out in text space) and the process (spread out in time) as trivialas possible.

In Dijkstra's case, it was the control structures that had to be dynamically inferred from the direct jumps written in the program text. The solution was what we nowadays refer to as structured programming. We have the same problem with our ongoing relationships, which we currently cannot encode and later read statically in the program text, but have to visualize evolving over time.

6 DERIVING HIGH LEVEL ARCHITECTURE

Grouping the three text fields into the `ui`, the three temperature variables into `memory-model` and the defaults database access into `persistence`, we arrive at the high-level architecture shown in Listing 15.

```
ui           =|= memory-model.
memory-model := persistence.
persistence  |= memory-model.
```

Figure 15: High Level Architecture of Temperature Converter

Unlike the previous listings in this section, Listing 15 is pseudo code, because we don't yet have the required grouping mechanism for constraints.

Using procedural abstraction to perform the grouping does not work, the problems discussed earlier also apply.

7 STORAGE COMBINATORS

Storage Combinators [24] are a composable implementation of In-Process-REST [20] and adaptation of the REST [11] architectural style to non-distributed settings. The REST verbs represent a storage-oriented API over an open-ended set of URIs and transports.

Storing and retrieving information is one of the primary uses of computers, in fact Richard Feynmann, in his *Lectures on Computation* [10], noted the following:

> One of the miseries of life is that everyone names everything a litte bit wrong, and so it makes everything a little harder to understand in the world than it would be if it were named

> differently. A computer does not primarily compute in the sense of doing arithmetic. Strange. Although they call them computers, that's not what they primarily do. They primarily are filing systems.

Storage combinators let us abstract over storage by having stores that can be composed directly similar to the way HTTP intermediaries such as caches, load-balancers etc. can be added to an HTTP processing chain.

A store is an object that implements the `Storage` protocol shown in Listing 16

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

Figure 16: Storage protocol expressed in Objective-Smalltalk

Example store endpoints include disk stores, dictionary stores, file-system stores and HTTP stores. Storage combinators 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.

Example storage combinators include a mapping store, which performs a user-defined mapping operation either on the data being loaded or stored or on the reference (URI) being used. A serialiser is a mapping store that serialises data being stored (for example to JSON) and de-serialises data being loaded. It is frequently combined with the disk or HTTP endpoints. A caching store maintains the relationship between a source store and a cache store.

With these stores, we can build a typical storage stack shown in Figure 17.

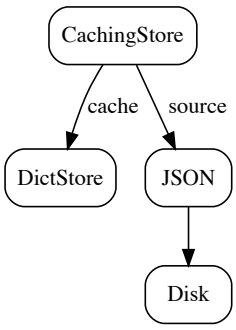

Figure 17: Common storage stack

This graphical depiction corresponds 1:1 to a textual configuration. For testing purposes, the stack can be replaced with just the dictionary store. Similarly, a simple dictionary based in-memory store for our temperature converter example can be replaced with this persistent store without impacting any of the rest of the code.

This storage stack has the disadvantage that writes are synchronous, therefore limiting performance. Making it asynchronous is shown in Figure 18.

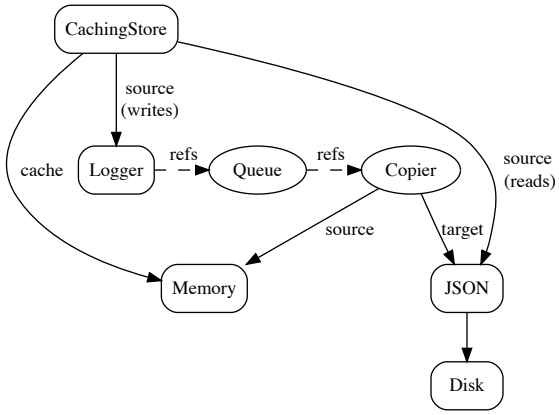

Figure 18: Asynchronous writer as a composition

This construction is based on the storage stack in Figure 17, but replaces the direct connection of the writing part of the `source` (disk) side of the cache with a reference to a logger. A logger is a store that logs the references and operations (GET, PUT, DELETE, ..) but not the data. For writes, instead of writing the data, the reference to the data is logged to a queue that feeds into a a copier, which. The copier copies the data from the in-memory cache where it was previously written to the its target, in this case the serialiser and disk store.

As before, the diagram is not conceptual but corresponds directly to the in-memory configuration, is in fact generated from it, and to a textual representation without additional glue code.

In real-world uses such as Wunderlist and Microsoft To-Do, significant application functionality was modelled using around a dozen stores, split evenly between application-specific stores and the kinds of generic stores shown here. Developers using stores reported code-reduction and productivity improvements of at least a factor of two.

The UI part of our temperature converter example is also handled by having UI elements store a reference to the element they represent. Changes in the model are recorded by a logger and broadcast so UI elements know to update themselves.

In Objective-Smalltalk, stores can be expressed directly as a kind of component akin to a class, as shown in Listing 19. The store (called a `scheme`) defines an instance variable `db`, two instance methods (with the '-' prefix) and two *property paths* with the '/' prefix.

Property paths allows the store to resolve arbitrarily nested paths of URIs with variable path components.

```
scheme SQLiteScheme  {
  var db.

  -initWithPath: dbPath {
     self setDb:(FMDatabase databaseWithPath:dbPath).
     self db open.
     self.
  }

  -dictionariesForQuery:query {
     self dictionariesForResultSet:(self db executeQuery:query).
  }

  /. {
     |= {
       resultSet := self dictionariesForQuery: 'select name from sqlite_master where [type] = "table" '.
       names := resultSet collect at:'name'.
       names := names, 'schema'.
       self listForNames:names.
     }
  }


  /schema/:table {
     |= {
        resultSet := self dictionariesForQuery: "PRAGMA table_info({table})".
            columns := resultSet collect: { :colDict |
            #ColumnInfo{
                                #name : (colDict at:'name') ,
                                #type : (colDict at:'type')
                        }.
        }.
        #TableInfo{ #name : table, #columns : columns }.
     }
  }
```

Figure 19: Part of a database adapter store in Objective-Smalltalk

8 DATAFLOW

In Infopipes [5], Andrew Black makes a point similar to Richard Feynman's, though he places communication at the center instead of storage:

> Recent years have witnessed a revolution in the way people use computers. In today's Internet-dominated computing environment, information exchange has replaced computation as the primary activity of most computers.

Infopipes are an object-oriented pipes and filters system similar to the one used in Objective-Smalltalk, which has been described in part in *iOS and macOS Performance Tuning* [19] as well as in *Standad Object Out: Streaming Objects with Polymorphic Write Streams* [23]. Key element is the `Streaming` protocol shown in Figure 20. It defines a single message, `writeObject:`, which takes a single argument.

A filter has a target that also conforms to the `Streaming` protocol and is therefore symmetric. The basic construction is passive and therefore does not require concurrency. However, the mechanism is asynchrony-agnostic, therefore asynchronous elements do not have to be handled specially, which is a tremendous benefit in code that has to deal with network requests: no call-backs, no pyramid-of-doom, no futures or promises or async-await. Just compose the filters according to what they are supposed to do and they will work in either synchronous or asynchronous environments.

```
@protocol Streaming

-(void)writeObject:anObject;

@end
```

Figure 20: Streaming Protocol

The base filter package is written in Objective-C, with a filter mapping onto an Objective-C class. Although some metaprogramming is used to reduce the ceremony required, there is still some overhead compared to defining a function or method. Objective-Smalltalk uses the syntax shown in Listing 21 to bring the overhead of defining a filter down to that of a method.

```
filter toupper
   |{  ^object stringValue uppercaseString. }
```

Figure 21: Filter definition in Objective-Smalltalk

The code defines a filter named `toupper`. This filter is defined by a single filter method, which gets added to the class that is created as `-writeObject:`. The argument of the filter method is automatically bound to the local variable `object` and the caret (ˆ) indicates the result, which in this case is written to the next filter in the pipeline, rather than returned to the caller as in Smalltalk.

8.1 Connect and Run

The filter defined above can be used as shown in Listing 22

```
(stdin -> toupper -> rawstdout ) run
```

Figure 22: Filter use Objective-Smalltalk

This "connect and run" idiom is common in architecture-oriented languages, as well as with the constraint connectors and storage combinators shown here. It might make a good replacement for and generalisation of "apply/eval".

9 RELATED WORK

Architecture Description Languages (ADLs) like ACME [7] or Rapide [16], as the name implies, *describe* architecture, they are not actually constructive. An exception is Unicon [26], which can generate code for the architecture described in the tool. However, it also is a separate language and mechanism, distinct from the substrate programming language. With all these systems, describing the architecture is in *addition* to programming the system. In Objective-Smalltalk, software architecture is actually used to simplify programming.

ArchJava [2] is the only other language the author is aware of that actually adds architectural elements to a general purpose programming language. However, it extends an full general purpose call/return programming language, whereas conceptually call/return is a special case. The ArchJava experience reports were one of the reasons for the creation of Objective-Smalltalk as a separate programming language rather than an extension of an existing language.

9.1 The "gentle tyranny"

While there are few attempts to generalise from call/return to connectors, there are many, trying to simulate different architectural styles using call/return style programming is very common.

For dataflow, examples include now-common stream processing frameworks such as Java Streams and so-called Functional Reactive Programming such as Rx, FlapJax, ReactiveCocoa and Combine. All of these simulate dataflow programming using call/return. Listing 23 shows an example of a Java Stream.

What is immediately noticeable is that this looks pretty much the way we would expect a natively expressed dataflow to look. So what is the problem? The problem is that the similarity is at best superficial, and even this superficial similarity is only obtained at a heavy cost.

```
myList
    .stream()
    .filter(s -> s.startsWith("c"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);
```

Figure 23: Java Stream

The code describes a pipeline that filters strings to those that start with the letter "c", converts those to upper case, sorts the list and finally outputs it to the system console. The pipeline character is suggested by the chaining of the method calls that indicate pipeline stages processing the data.

However, the method calls that are chained do not, in fact, process any data, even though that is what is suggested by the code as written. Instead they are a fluent interface for *constructing* filter objects that will then process the data. Because we cannot write down a chain of filter objects directly, we have to introduce and maintain an extra layer of indirection in order to make the code look "natural".

In part due to this indirection, adding filters is a sufficiently complex task that it is not intended for users of the streams framework. In addition to creating the filter, users also have to add a method to the builder(s), which requires either a form of class extensions or modifications to a library class.

The difficulty of extending the set of filters means custom processing has to be performed by anonymous functions inside of `map` or `filter` filters, which makes it hard to extract and encapsulate functionality in named components.

Balanced against these difficulties is the fact that streams that are implemented in this way integrate with the rest of the call/return based environment, sort of, and look like what we're trying to accomplish, sort of.

And this is why the tyranny of the call/return architectural style exerts its force *gently*: it is so flexible that we can get so close to other architectural styles that pointing out the differences appears to be splitting hairs.

However, the differences are not small, just subtle.

10 SUMMARY AND OUTLOOK

One of the reasons programming is so hard and requires seemingly excessive amounts of engineering is that the (linguistic) tools we use no longer match the systems we are expected to build using those tools.

However, the assumption that this particular architectural style is the only one that amounts to "programming" is so deeply entrenched that we tend to describe alternatives as *not-programming*, so *modeling*, *configuring* or architectural-*description*.

The Objective-Smalltalk project is an exploration of the hypothesis that we should be able to *program* using alternative architectural styles on an equal footing with call/return. It does *not* call for replacing call/return, but instead for generalising it.

Specific styles that have been explored include constraint connectors, in-process REST with Storage Combinators and in-process pipes and filters. All show significant potential for reducing accidental complexity and some have already proven that potential in successful real world projects.

While more research is needed both on the meta-architecture and the specific styles included, Objective-Smalltalk and its bundled styles are ready to move from experimentation and validation to (early) adoption.

REFERENCES

[1] 2002. *Resolved: Objects Have Failed.* ACM. http://www.dreamsongs.com/ObjectsHaveFailedNarrative.html

[2] Jonathan Aldrich, Craig Chambers, and David Notkin. 2002. ArchJava: Connecting Software Architecture to Implementation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*. Association for Computing Machinery, New York, NY, USA, 187–197. https://doi.org/10.1145/581339.581365

[3] Apple Inc. 2015. What Are Cocoa Bindings? https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CocoaBindings/Concepts/WhatAreBindings.html

[4] Andrew P. Black. 2013. Object-oriented programming: Some history, and challenges for the next fifty years. *Information and Computation* 231 (2013), 3 – 20. https://doi.org/10.1016/j.ic.2013.08.002 Fundamentals of Computation Theory.

[5] Andrew P. Black, Jie Huang, Rainer Koster, Jonathan Walpole, and Calton Pu. [n. d.]. Infopipes: An abstraction for multimedia streaming. *Multimedia Systems* 8, 5 ([n. d.]), 406–419. https://doi.org/10.1007/s005300200062

[6] Stéphane Chatty. 2008. Programs = Data + Algorithms + Architecture: Consequences for Interactive Software Engineering. In *Engineering Interactive Systems*, Jan Gulliksen, Morton Borup Harning, Philippe Palanque, Gerrit C. Veer, and Janet Wesson (Eds.). Springer-Verlag, Berlin, Heidelberg, 356–373. https://doi.org/10.1007/978-3-540-92698-6_22

[7] David Wilkie David Garlan, Robert Monroe. 1997. *ACME: An Architecture Description Interchange Language.* Technical Report. Carnegie Mellon University.

[8] Edsger W. Dijkstra. [n. d.]. Letters to the Editor: Go to Statement Considered Harmful. *Communication of the ACM* 11, 3 ([n. d.]), 147–148. https://doi.org/10.1145/362929.362947

[9] Stuart Feldman. 2004. A Conversation with Alan Kay. *Queue* 2, 9 (Dec. 2004), 20–30. https://doi.org/10.1145/1039511.1039523

[10] Richard P. Feynman. [n. d.]. *Feynman Lectures on Computation.* Addison-Wesley, Boston, MA, USA.

[11] Roy Thomas Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures.* Ph.D. Dissertation. University of California, Irvine.

[12] 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

[13] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[14] Glenn E. Krasner and Stephen T. Pope. 1988. A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80. *J. Object Oriented Program.* 1, 3 (Aug. 1988), 26–49.

[15] Trygve M. H. Reenskaug. 1979. Thing-Model-View-Editor – an Example from a planningsystem. 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

[16] Rapide Design Team. 1997. *Guide to the Rapide 1.0 Language Reference Manuals.* Technical Report. Stanford University.

[17] Mary Shaw and David Garlan. 1996. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall.

[18] Marcel Weiher. 2010. Objective-Smalltalk: . http://objective.st

[19] Marcel Weiher. 2017. *iOS and macOS Performance Tuning: Cocoa, Cocoa Touch, Objective-C, and Swift.* Addison-Wesley Professional.

[20] Marcel Weiher and Craig Dowie. 2014. *In-Process REST at the BBC.* Springer New York, New York, NY, 193–209. https://doi.org/10.1007/978-1-4614-9299-3_11

[21] 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)*. Association for Computing Machinery, New York, NY, USA, 61–72. https://doi.org/10.1145/2508168.2508169

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

[23] Marcel Weiher and Robert Hirschfeld. 2019. Standard Object out: Streaming Objects with Polymorphic Write Streams. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages (DLS 2019)*. Association for Computing Machinery, New York, NY, USA, 104–116. https://doi.org/10.1145/3359619.3359748

[24] 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! 2019)*. Association for Computing Machinery, New York, NY, USA, 111–127. https://doi.org/10.1145/3359591.3359729

[25] David J. Wheeler. 1952. The Use of Sub-routines in Programmes. In *Proceedings of the 1952 ACM National Meeting (Pittsburgh) (ACM '52)*. ACM, New York, NY, USA, 235–236. https://doi.org/10.1145/609784.609816

[26] Gregory Zelesnik. 1992. *The UniCon Language Reference Manual.* Technical Report. Carnegie Mellon University.