



Beyond Procedure Calls as Component Glue

Connectors Deserve Metaclass Status

Marcel Weiher

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

Marcel Taeumel

marcel.taeumel@hpi.de
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany

Robert Hirschfeld

robert.hirschfeld@uni-potsdam.de
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany

Abstract

We present the architecture-oriented programming language Objective-S, which goes beyond procedure calls for expressing inter-component connectors (so-called glue code) in order to directly express a wide range of architectural patterns directly in the implementation.

Previous approaches for encoding architecture require either indirection, maintaining separate architectural descriptions, or both. Expressing the architecture directly in the implementation instead of indirectly avoids the problems associated with creating and maintaining duplicate representations.

Objective-S introduces syntactic elements that let us express many architectural connections directly using a simple surface syntax. These surface elements are supported by a metaobject protocol of polymorphic connectors.

The key insight of our approach is that while so-called general-purpose programming languages do lack the ability to express most architectural concerns directly, as shown by previous research, this is not an inherent limitation.

With Objective-S, we take connectors that already exist in implementation languages, such as data access or procedure calls and make them polymorphic using a metaobject protocol. This metaobject protocol enables polymorphic connection using glue code at the metaobject level that is both generic and mostly hidden from the surface language.

CCS Concepts: • Software and its engineering → General programming languages.

Keywords: Architecture, Metaprogramming, Language, REST

ACM Reference Format:

Marcel Weiher, Marcel Taeumel, and Robert Hirschfeld. 2024. Beyond Procedure Calls as Component Glue: Connectors Deserve

Metaclass Status. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '24)*, October 23–25, 2024, Pasadena, CA, USA. ACM, New York, NY, USA, 17 pages. <https://doi.org/10.1145/3689492.3690052>

1 Introduction

As software systems have gotten larger, more diverse and increasingly constructed from existing components, the way we connect (“glue together”) those components has become increasingly important. In the field of software architecture that glue is known as *connectors*, whereas the loci of computation and storage are known as *components*.

The overall architecture is determined by the *connectors* that connect these components together, for example by mediating their communication in order to create complete computational systems [21]. The connectors are the arrows in typical “boxes and arrows” informal architectural diagrams (see Figure 1).

Current general purpose programming languages are more focused on expressing the algorithms and data structures that make up the components. They have very limited facilities for expressing the connections between components. With a few exceptions, some variant of the procedure call [24] is the only connector provided that allows defining custom connections. Other kinds of connections are achieved by calling procedures, both for setting up the connection and for implementing the communication over the connection.

As an example, a Unix pipeline is set up by first calling the `pipe()` system call, then creating one or more subprocesses using `fork()`, and actually communication is performed using the `read()` and `write()` system calls, with all the system calls exposed as procedures (called *functions* in C).

In the shell, a Unix pipe is set up by using the vertical bar symbol between two commands: `a | b`. This is a direct expression, but requires using a domain specific language and is limited to a specific kind of connector.

Proposed mechanisms for achieving direct expression of architectural concerns have centered around giving first class status to connectors [18]. They have ranged from early module interconnection languages, coordination languages, architecture description languages (ADLs) and even programming languages augmented with a connector abstraction [1].



This work is licensed under a Creative Commons Attribution 4.0 International License.

Onward! '24, October 23–25, 2024, Pasadena, CA, USA

© 2024 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-1215-9/24/10

<https://doi.org/10.1145/3689492.3690052>

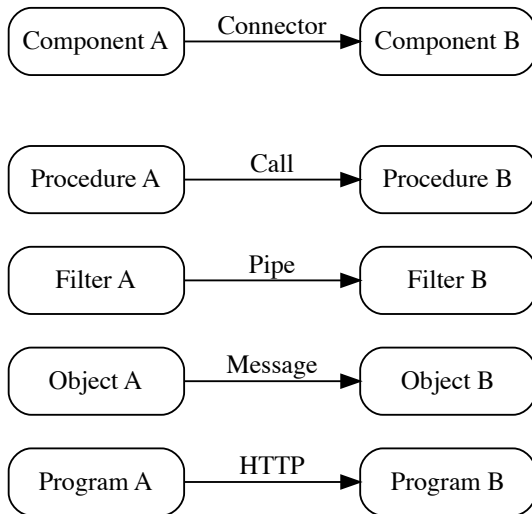


Figure 1. Boxes and Lines for Components and Connectors

Table 1. Connector classes and variants, adapted from [15]

Connector Type	Variants
Procedure Call	subroutine, function, method call, macro, coroutine, system call ...
Data Access	Local variable, file API, REST APIs, ...
Stream	Unix pipes, Go channels, TCP sockets, ...
Event	notifications, message brokers,..
Distributor	naming, DNS, X-400,..
Linkage	compile-time, run-time, dynamic, static
Adaptor	Interpreters, DeLine packaging,..
Arbitrator	locks, transactions,..

Having two separate representations comes with the drawbacks of creating, maintaining and synchronizing these two representations. Having a single representation would avoid these drawbacks.

Objective-S makes it possible to express software architecture directly in the implementation, to program directly with software architectural abstractions, by extending the supported connector types beyond the ones supported in general purpose programming languages.

Previous research [15][19] has identified and categorized connectors, summarized in table 1. shows connectors grouped into general connector types (first column) and then a few of the variants within that general type (second column).

Most general-purpose programming languages have direct support for a specific subset of the connector variants found in the second column of Table 1, usually a procedure call and variable access, but cannot directly express other variants or connector types.

Those languages can be said to be architecturally monomorphic, that is they only directly support at most one variant

of a specific connector type. A language would be architecturally polymorphic if it were able to directly express at least several, better, yet most or all of the variants of a given connector type.

Our approach with Objective-S is first to make existing connectors polymorphic, so they are generalized from a single connector variant to an entire connector type, while keeping the surface syntax close as close to that of a single variant. This is done by having an extensible object-oriented framework of architectural abstractions that can be integrated into the language using a fairly narrow metaobject protocol.

Second, we minimally extend the surface language to support additional connector types. Finally, we add a generic mechanism for expressing component connection and add some supporting mechanisms such as complex object literals.

The rest of the paper will start by introducing the architectural toolkit and language features that make up Objective-S, with section 2 demonstrating how the supported connectors are made polymorphic. Section 3 details the polymorphic connection operator (\rightarrow), and section 4 introduces some auxiliary features. Section 5 demonstrates how a variety of connector variants and architectural styles can be expressed using these polymorphic connectors and combinations thereof. Section 6 shows frameworks and applications built with Objective-S. Section 7 discusses and evaluates the results. Section 8 puts this work into context of previous publications and section 9 summarizes the findings and points to future work.

2 Polymorphic Connectors in Objective-S

Objective-S is an architecture-oriented programming language implemented as a generalization of Smalltalk [10] and implemented in Objective-C [6]. Objective-S is a full peer to Objective-C, so in the spirit of reusing existing components without additional glue, messages can be sent from Objective-C code to Objective-S code and vice versa without a FFI or other bridging mechanism. Methods in either language are essentially indistinguishable for callers.

Objective-S directly supports the first 5 connector types listed in Table 1: procedure call, data access, stream processing, event processing and naming. For each of these connector types, it provides a common syntax for use at the language level, an API (the metaobject protocol) for plugging concrete instances into the language, and different ways of selecting which instance to use. The linkage and adaptor connectors are supported indirectly.

A framework of concrete connectors and components that can be plugged into the metaobject protocol is also provided.

2.1 Polymorphic Identifiers / Storage Combinators

Polymorphic identifiers (PIs) [27] implement the naming connector type and connect to the underlying storage that

implements the data access connector type provided by storage combinators [30]. Syntactically, PIs are derived from and closely follow URI syntax. Like other programming languages, read access is expressed by writing the PI in an expression and write access is expressed by placing the PI on the left hand side of an assignment expression, with the assignment using either the Pascal/Smalltalk convention (“:=”) or a left-facing arrow (“←”). Listing 1 shows a few sample assignments.

```
greeting := 'Hello World!'.
var:greeting := 'Hello World!'.
file:hello.txt := greeting.
var:conf/2024/website := https://2024.splashcon.org.
```

Listing 1. Polymorphic Identifiers

Identifiers without a scheme are prepended with the current default scheme. In the example above, the default scheme is `var:`, so `var:greeting` and `greeting` resolve to the same reference.

Data access is mediated by the storage combinator metaobjects via the Storage protocol shown in Listing 2.

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

Listing 2. Storage Protocol

Data retrieval, when an identifier is mentioned in an expression, is handled by the `at:` message, data storage, when an identifier is at on the left hand side of an assignment, by `at:put:.` The `id` parameters are the polymorphic identifiers that are passed from the language to the respective stores.

In the following text, we will use URI and Polymorphic Identifier interchangeably.

2.1.1 Polymorphic References. In Objective-S identifiers denote values, so `file:hello.txt` refers to the contents of the file `hello.txt`, and `hello` refers to the contents of the variable `hello`.

To get a reference (pointer) to a value, the special scheme `ref:` is prepended to the identifier: `ref:file:hello.txt` is a reference to the file and `ref:hello` is a reference to the variable `hello`.

References are created by stores by passing them a polymorphic identifier. The store controls what kind of reference to return to the client. The default implementation stores the original identifier and the store,

2.1.2 Property Paths. Property paths extend the concept of properties (which formalize the pairing of accessor methods to Load Update Pairs (LUP) [23]), to allow access methods

to be defined not just for simple property names, but for complex paths. This allows direct specification of methods that handle entire classes of access paths.

Listing 3 shows the definition of a store for the tasks of a to do list application using property paths.

```
scheme Todo : MappingStore {
  var tasks.
  -taskList {
    this:tasks allValues.
  }
  /tasks {
    get { self tasks. }
  }
  /task/:id {
    get { this:tasks at:id . }
    put { this:tasks at:id put:newValue.
          self persist. }
  }
  /complete/:id {
    post { (this:tasks at:id) setDone:true.
           self persist. }
  }
  -<void>persist { source:tasks := self tasks. }
}
```

Listing 3. Todo store definition

Property paths are defined at the same level as methods, using the a leading slash (“/”) to indicate that what follows is a property path definition, followed by the path pattern to match. Inside the property path definition, the REST verbs `get`, `put`, `post` and `delete` can be given a method implementation. If the operation has a parameter, such as with a `put` or `post`, this is made available within the method using the name `'newValue'`. The path definition can contain path parameters (`:parameter`) and wildcards (`*`), with any named path parameters also available as bound variables in the method.

Paths are matched in the order they appear in the file. Property paths support inheritance, so a path not matched will be passed to the superclass. Only verbs mentioned are matched.

2.2 Streaming

Syntactic support for streaming consists of the `<<` operator for writing objects to a stream, taken from C++, with the special case of passing data to the next filter in a pipeline designated by the `^` operator.

```
stdout << 'Hello World!\n'.
```

Listing 4. Writing to a stream

The metaobjects implementing the stream abstraction are filter classes that implement the streaming protocol taken from Polymorphic Write Streams [29] and shown in listing 5.

```
protocol Streaming {
```

```
-<void>writeObject:anObject;
}
```

Listing 5. Storage Protocol

Any object that implements the Streaming protocol can be a stream target, any object that sends this message can act as a stream source. The protocol is widely adopted in Objective-S: in addition to many filters, polymorphic references, one-argument closures, and mutable arrays can also act as stream targets.

2.2.1 Defining Filters. Filters are objects under the hood, and so defining a new filter requires creating a new class. Objective-S includes syntax to bring the syntactic overhead of creating a filter down to roughly the same level as defining a method.

Listing 6 shows the definition of four filters, one that adds 3 to a number and one that multiplies by 4, one that converts a string to upper case and one that just duplicates the input.

```
filter add3 |{ ^object + 3. }
filter times4 |{ ^object * 4. }
filter upcase |{ ^object uppercaseString. }
filter dup |{ ^object. ^object. }
```

Listing 6. Example filters

Each filter definition creates a new class that is a subclass of the `Filter` class. The `|{` is shorthand for the definition of a `-<void>writeObject:object {` method that is the standard message with which filters communicate data.

2.2.2 Pipelines. Filters are connected into pipelines using the polymorphic connection designator (section 3). Listing 7 creates a pipeline of two filters that first adds 3 to a number passed in and then multiplies the result by 4.

```
add3 → times4
```

Listing 7. Pipe that adds and multiplies

2.2.3 Dataflow Constraints. Objective-S syntactically supports dataflow constraints using a *permanent assignment* operator: `|=`. The syntax is similar to the assignment syntax `:=`, because they do very similar things: assign the value on the right hand side to the location on the left hand side, with the difference that regular assignment does this once, whereas permanent assignment maintains a permanent connection and will update the left hand side whenever the right hand side changes.

Dataflow constraints are built using a combination of storage combinators and polymorphic write streams. The `LoggingStore` combinator acts as a decorator for any other store. It sends the references of any state update messages to a polymorphic write stream.

There is therefore no additional metaobject protocol for dataflow constraints.

2.3 Message Passing and Higher Order Messaging

The variant of the *procedure call* connector directly supported by Smalltalk is synchronous, late bound messaging, which is polymorphic in type but monomorphic in messaging semantics. Objective-S makes the messaging semantics polymorphic using Higher Order Messaging (HOM) [26].

A Higher Order Message consists of a prefix message and an argument message, or main message. The main message carries the intent and the prefix message specifies how the message should be delivered. Listing 23 gives an example:

```
robot async turn:20.
```

Listing 8. Async higher order message

Here, the robot is sent the `turn:20` message asynchronously.

Conceptually, all message sends are mediated by HOMs, with the HOM sent to the reference and then controlling how the argument message is sent to the object referred to by the reference. Similar to PIs, defaults recover the common behavior, with the default HOM being `sync`, a HOM requesting default Smalltalk synchronous dynamic message delivery.

3 Polymorphic Connection

So far, we’ve looked mostly at how to interact with new connectors and their associated components, but connecting components has been implicit and specific: the procedure call connection is established by simply mentioning the other procedure, a Smalltalk message send by mentioning the receiver object and the message name.

Having this sort of optimized syntax for connectors is crucial to making programming languages usable, but of course it usually only applies to a very narrow set of connectors.

Objective-S supports the right arrow (both `'→'` and ASCII `'->'`) as a polymorphic connection designator. It indicates that whatever is on the left hand side should be connected to whatever is on the right hand side, if possible.

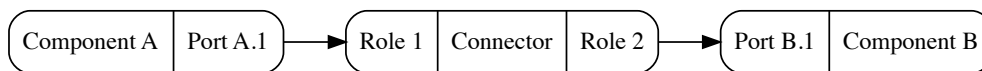
We call the `->` a “designator” and not an operator, because it does not indicate an imperative command to create the connection, to be executed once. Instead it is the syntactic element that defines the connection and does so for the lifetime of the connection. For example, a breakpoint set on a specific connection would trigger whenever communication occurs via that particular connection.

3.1 Ports and Roles

In order to explain how polymorphic connection works, we first need to briefly introduce a slightly more complex and complete view of connection than the one shown in figure 1. Figure 2 shows component connection via ports and roles as defined in the software architecture literature:

Interaction between components is still mediated via connectors. However, these connectors are now complex, with

Figure 2. Components with Ports, Connectors with Roles



roles as actual attachment points. Similarly, components now have ports as attachment points.

Due to syntactic constraints the Objective-S connection designator currently supports a limited version of this model, with the connector only having two fixed roles: left-hand-side (LHS) and right-hand-side (RHS).

3.2 Matching

A fully qualified connection definition has to specify the source component, the port of the source component to connect, the connector in question, the role of the connector that the source port connects to, the target component, the port of the target component to connect to and finally the role of the connector that the target port is connected to.

The polymorphic connection designator can be used to express such a fully qualified connection definition if necessary, but most of the time it suffices to just specify two components to be connected, leaving out the ports, roles and even the specific connector to be used.

When a connection between two components is requested without any further qualification, the matching algorithm first determines the default ports of the two components and then requests the default connector from those ports. It then matches the two ports to the roles of the connector and asks the connector to connect them by sending it a connect message.

A port can be specified using the port scheme. Listing 9 shows the `celsiusTextField` being connected to the port `setCelsius:` of the `model` object.

```
celsiusField → port:model/setCelsius: .
```

Listing 9. Connecting to a port

The port in question is a message port, the `celsiusField` will send the `setCelsius:` message to the `model` object with its current value as the argument when the user makes a change. Similar to CLIC [3], Objective-S makes all the messages an object responds to available as ports, by default.

3.3 Metaobject Protocol

To carry out this task of matching and connecting components via connectors, the Objective-S metaobject protocol has interfaces that represent components, ports and connectors. Just like objects conforming to the `Storage` protocol can

participate in variable retrieval and assignment operations, objects that conform to these protocol can participate in the connection matching process.

4 Support Features: Classes, Literals and Templates

In order to connect components we must create those components. Objective-S supports fairly conventional class definition syntax, with component definition as a minor extension.

Slightly more distinct are complex object literals, which allow writing the definition of arbitrary object instances in the program text without resorting to procedural creation.

4.1 Class and Component Definition

Unlike most Smalltalks, Objective-S has syntax for class and method definitions. Listing 10 defines a single class `Task` with three instance variables and a single method.

```
class Task {
    var id.
    var <bool> done.
    var title.
}
-description {"Task {this:title} done: {this:done}"}
```

Listing 10. Tasks class

An analogous syntax also defines other kinds of components such as filters or scheme handlers, as well as some user-level connectors such as protocols. Component definitions are mapped onto class definitions.

Listing 11 shows the definition of the `ModelDidChange` protocol:

```
protocol ModelDidChange {
  -<void>modelDidChange:object;
}
```

Listing 11. Protocol definition

A protocol corresponds to an interface in languages like Java.

4.2 Complex Object Literals

In order to avoid “accidentally algorithmic” code, Objective-S includes a facility for writing complex objects as literals directly in the program text without having to write procedural code that constructs them.

Program text usually describe classes, whereas systems typically consist of connected object instances. This dichotomy leads to indirection, with object instance having to be defined and connected using procedural code, leading to exactly the kind of indirection Objective-S is trying to remove [2]:

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.

Complex object literals are closely modeled on the syntax for object and array literals in other languages, particularly JSON. Like JSON, arrays are delimited by square brackets and dictionaries by curly brackets. However, partly in order to distinguish them from closures, dictionaries need to have their opening curly bracket preceded by a hash mark.

An optional class name may be provided between the hash mark and the opening curly brace. If it is given, an instance of that class is constructed. If not, a dictionary is constructed. Listing 12 shows the definition of a single Task instance.

```
#Task{ id:0, title: 'Submit paper', done: false }
```

Listing 12. Tasks data

Arrays do not require the leading hash mark, but can use one to construct an object that can be created from an array. Listing 13 defines an array of two Task instances, one completed, one not.

```
[ #Task{ id:0, title: 'Submit paper', done: false },
  #Task{ id:1, title: 'Write code', done: true } ].
```

Listing 13. Tasks data

5 Polymorphic Connectors for Architectural Styles

The previous sections gave an overview over the language features that enable polymorphic connectors and connection, along with a set of connectors with those features.

This section uses these capabilities to express specific architectural patterns directly rather than indirectly, without additional procedural glue code at the point of connection.

We also show how this feature of direct connection is robust in the face of connector variations of kind and of scale.

The uniformity of concrete connectors within a connector type makes composition via the connection designator possible, and the architecture visible.

5.1 Dataflow

Connecting user interface elements to the rest of the program does not naturally match the call/return architectural style: neither is the user interface element naturally a subroutine of the client that employs it, nor is the client naturally a subroutine of the user-interface element that computes some value for that element and then returns control to it.

Overcoming this mismatch usually requires some sort of procedural callback, the Inversion of Control architectural pattern.

For example, if we wanted to print user input from a text field to the console, this would be mean connecting the text field to stdout. With procedural code, this requires installing some sort of callback, such as shown in listing 14.

```
text := #TextField{ frame:(180@24) }.
text setCallback: { :text | stdout println:text. }.
```

Listing 14. UI to stream

The text field with width 180 and height 24 is visible enough thanks to complex object literal syntax, but the target is almost entirely obscured by the glue code, in fact it’s not even clear that there are elements that are being connected.

In Objective-S, stdout is a Polymorphic Write Stream and UI elements are stream sources, so the connection can be made directly. Listing 15 shows the same text field connected to Unix stdout.

```
text := #TextField{ frame:(180@24) }.
text → stdout.
```

Listing 15. UI to stream

The connection is clearly visible and the only glue is the the generic connection designator “→”.

This clear visibility of the architecture, of the components and their connections, remains if we make the example more complex by adding custom processing.

Listing 16 uses the upcase filter from listing 6 as a stand-in for custom processing to be done before sending the result to stdout.

```
upout := upcase → stdout.

text := #TextField{ frame:(180@24) }.
text → upout.
```

Listing 16. UI to stream

After defining the named filter, which could have been done elsewhere, the actual connection code is exactly the same.

In the callback case, we would probably modify the code as shown in listing 17.

```
text := #TextField{ frame:(180@24) }.
text setCallback: { :text | stdout println:text
  uppercaseString. }.
```

Listing 17. UI to stream

This obvious extension of the previous code from listing 14 exposes a problem that was already present before, but not yet apparent: we are performing actual computation in what should only be glue code, so mixing up view and model.

Breaking Model/View separation – mixing model code into what should be only view code – is an extremely common problem in UI programming and one major difficulty with the MVC pattern in industry.

To fix it, what we should have done from the start is create a model object that handles the actual action that is to be triggered by the UI, as shown in listing 18.

```
class Model {
  -<void>printText:text {
    stdout println: text uppercaseString.
  }
}
model := Model new.
text := #TextField{ frame:(180@24) }.
text setCallback: { :text | model printText:text. }.
```

Listing 18. UI to stream

However, that seems like a lot of effort for a gain that at least initially seems mostly theoretical. The underlying problem is that our connection code is “accidentally algorithmic”: it is using procedural abstraction, best suited for computation, to create a connection.

Polymorphic references (section 2.1.1) are also integrated as stream sinks or sources, so listing 19 will show a text field and overwrite the contents of the file /tmp/hello.txt with whatever is typed in the field.

```
text := #TextField{ frame:(180@24) }.
text → ref:file:/tmp/hello.txt.
```

Listing 19. UI to file (variable)

Single argument blocks also act as filters, and can thus be used in situations where a stream is expected. So if putting custom code directly inline in the glue specification is required, it is also possible, as seen in listing 20.

```
text := #TextField{ frame:(180@24) }.
text → { stdout println:$0. }.
```

Listing 20. UI to closure

Of course, dataflow is not just useful for connecting UIs. For example the Objective-S standard library provides XML and JSON parsing and generation services as composable filters. Creating a Unix filter from an Objective-S filter is also

straightforward, listing 21 shows how the upcase filter from listing 6 is packaged as a Unix filter.

```
#!env st
filter upcase |{ ^object uppercaseString. }
(stdin → upcase → rawstdout ) run
```

Listing 21. UI to closure

Similarly, Unix filters are adapted to Objective-S filters where needed.

5.2 Call/Return and Objects and Messages

Objective-S supports the call/return architectural style using Smalltalk-style messaging. This specific instance of the procedure call connector type from Table 1 is synchronous, dynamically dispatched, local and has a single receiver. Figure 22 shows such a message.

```
robot turn: 20.
```

Listing 22. Standard message send

This could also be expressed using a fully qualified HOM as `robot sync turn:20`, but with the default HOM being `sync`, this can be omitted. All these characteristics can be altered via a Higher Order Message. Figure 23 shows the same message still with a single local receiver, dynamically dispatched, but dispatched asynchronously.

```
robot async turn: 20.
```

Listing 23. Asynchronous message send

Sending the same `turn: message` asynchronously with Cocoa APIs requires turning the message to be sent into arguments of an asynchronous dispatch message, as shown in listing 24

```
receiver performSelectorInBackground: #turn:
  withArgument: 20.
```

Listing 24. Asynchronous message without HOM

Further expanding the scope of messaging to other processes or even remote systems typically requires completely abandoning the language-provided messaging for a reified messaging mechanism such as Mach messages or, in listing 25, BMessages of the Be/Haiku operating system.

```
BMessage *msg = new BMessage(WINDOW_REGISTRY_SUB);
msg->addFloat( "degreesToTurn" , 20.0 );
messenger.SendMessage(msg);
```

Listing 25. BMessage robot turning

With a HOM, as used by the Croquet system [22] for this purpose, even a remote message reuses the familiar syntax, as seen in listing 26:

```
robot future turn: 20.
```

Listing 26. Asynchronous message

Table 2. Data access variants

Data Store	Convention PL	Objective-S
Local variable	hello	hello
Environment variable	getenv("hello");	env:hello
Dictionary	dict at: 'hello'	dict:hello
File	f=fopen("hello","r"); fread(buffer,len,f);	file:hello
Keychain	SecItemCopyMatching(queryDict , &retval);	keychain:hello
Dynamic Library	dlopen("hello.dylib", RTLD_NOW);	dl:hello
Web	urllib.urlopen("http://hello.com")	http://hello.com/

We can also alter the “single receiver” dimension, by sending the message to an entire collection of objects with the `do` Higher Order Message, as shown in listing 27

```
robots do turn:20.
```

Listing 27. Multiple receiver message

Previous work on Higher Order Messages used the message intercession capabilities of dynamic object oriented languages, so being less dynamic than standard messaging was not possible.

With HOMs being supported directly by the compiler, Objective-S removes this limitation and thus makes messaging more completely architecturally polymorphic. Static dispatch now also becomes possible. Listing 28 shows the syntax for static binding and inlining.

```
receiver static msg:argument.
receiver inline msg:argument.
```

Listing 28. Static dispatch and inlining via HOM

5.3 (In-process) REST

The REST architectural style [9] underpins the World Wide Web. In-Process REST [25] was the result of the realization that many of the qualities of this data-access-oriented, distributed architectural style are also beneficial in a non-distributed context.

Table 2 gives a small example of how polymorphic identifiers and the store abstraction treat the data access connector uniformly for a wide variety of data stores at a wide range of scales. The uniformity actually goes further than the *Objective-S* columns suggests: the scheme names are user-definable, so they can also be made to match exactly.

The uniformity also goes beyond the syntax: the data-access metaobject protocol for all these stores is also the same, meaning that clients written for one store usually work for all other stores as well. As an example, the URI `file:.` will get a directory listing in the `st`, the Objective-S interactive shell. As listing 29 shows, the same mechanism also works with environment variables or instance variables of a Task object.

```
] file:.
ConnectorHierarchy.pdf      connectors-metaclass.aux
connectors-metaclass.bb1   connectors-metaclass.bib
connectors-metaclass.blg   connectors-metaclass.log
connectors-metaclass.out   connectors-metaclass.pdf
connectors-metaclass.synctex.gz connectors-metaclass.tex
] env:.
__CFBundleIdentifier      TMPDIR      XPC_FLAGS
TERM                      DISPLAY     SSH_AUTH_SOCK
TERM_SESSION_ID          SHELL      HOME
LOGNAME                   USER      PATH
] task:.
isa id done title
```

Listing 29. Directory listing for file:, env: and a task

```
scheme:s3 := ref:http://localhost:9000/ asScheme
scheme:s3 source setHeaders: #{ user: 'minioadmin' , password: 'minioadmin' }.
text → ref:s3:bucket1/message.txt.
```

Listing 30. S3 scheme defined and used

Listing 31 shows how the To Do store defined in listing 3 can be turned into an actual system by composing it with a file-store via a JSON translator.

```
fromJSON := #JSONConverter{ up: false, converterClass:
class:Task }.
todo := #Todo{ tasks: (tasks dictionaryByKey:'id'.) }.
```

```
todo → fromJSON → ref:file:/tmp/tasks/ asScheme.
```

Listing 31. Todo system

The JSONConverter is a storage combinator that translates objects between its source and destination from or to JSON using JSON converter streams.

The top level structure of the system is seen in the last line of code: the `todo` store is connected to the file-system via the JSON converter.

5.4 Events / Implicit Invocation

Event programming is similar to messaging, but without the sender specifying the receiver(s) of the message. Instead, the message is broadcast, usually via some sort of channel abstraction, and receivers can register their interest by subscribing to specific channels.

In this way, senders and receivers are more highly decoupled.

5.4.1 Local Events. The implicit invocation style is provided via notification protocols. A notification protocol defines a single message that is sent to objects that subscribe to the protocol when a notification is sent.

The notification is sent by sending the `notify: message` to the protocol object itself, as seen in Listing 33.

```
protocol:ModelDidChange notify: ref:myObject.
```

Listing 32. Sending a notification via protocol

An adapter to the polymorphic write streams sends the notification message on every object written to the stream.

```
protocol:ModelDidChange << ref:myObject.
```

Listing 33. Sending a notification via protocol

Objects can subscribe and unsubscribe to notifications manually if so desired. More simply, a class can adopt the notification protocol and its objects will then be subscribed automatically.

Among other things these notifications are used to handle Model → View communication in our implementation of the MVC pattern. Views simply adopt the `ModelDidChange` protocol. Stores used by the model use a logging store to send the `ModelDidChange` notification. This notification then gets translated into sending the view a `modelDidChange: message` with a reference to the object that changed so it can refresh itself.

5.4.2 Distributed Events. Event processing is widely used to glue together distributed systems, often asynchronously. There are a wide variety of systems, such as Amazon SQS, Microsoft Event Bus, Kafka, RabbitMQ, ZeroMQ etc.

These systems are also often referred to as messaging or streaming services, and they combine some of the features of both messaging and streaming. In Objective-S, we model distributed event systems using the stream abstraction.

Listing 34 shows how to send an event to a RabbitMQ¹ queue on the local host.

```
r := #OQRabbit{ queue:'testQueue' , sending: true ,
  host:'localhost' }.
r << 'Hello Objective Rabbit '.
```

Listing 34. Sending an event to a RabbitMQ queue

This code and the other examples omit some setup details by relying on defaults for exchange configuration, channel setup and credentials. Listing 35 connects a text field directly to a queue.

```
queue := #OQRabbit{queue:'testQueue' , sending: true}.
text := #TextField{ frame:(180@24) }.
text → queue.
```

Listing 35. Connecting UI to a RabbitMQ queues

Receiving from a queue is very similar. Listing 36 connects a local queue to stdout, so incoming events are logged to the console.

```
queue := #OQRabbit{ queue:'testQueue' , host:'
  localhost' }
queue → stdout
```

Listing 36. Receiving events from a RabbitMQ queue to stdout

The same receive/logging functionality using the Python client libraries is shown in Listing 37. As with the Objective-S examples, some configuration details have been omitted, the actual example is 52 lines of code vs. the 12 shown here.

Although the difference in pure code bulk is also noticeable, for us the more significant factor is how the architecture is obscured. The actual destination is hidden in the 2nd line, implicit in the print statement of the function that is the callback given to the channel in the second to last line.

The remote event stream can be connected to any other stream-compatible target. Listing 38 shows a queue being connected to the `ModelDidChange` protocol notification, so events coming in on this particular queue trigger the UI to be updated.

```
queue := #OQRabbit{ queue:'testQueue' }
queue → protocol:ModelDidChange.
```

Listing 38. Distributed `ModelDidChange` notifications

This can be used to update a local UI when some remote resource changes, a remote observer pattern.

5.5 Observer

The Observer pattern is implemented by the *Log* storage combinator, which has a base store and a Polymorphic Write Stream that is the log itself. Listing 39 shows a conceptual implementation of the logging store.

```
scheme Log {
  var stream.
  /*:uri { put { source at:uri put:value. stream <<
    `uri. } }
}
```

Listing 39. Logging store

With the wildcard, the property path matches all writes. It passes the write on to its source store and then logs the URI to the stream attached to its logging port.

Any filter attached the logging port is thus notified of all writes. Connecting the logging stream to the adapter for protocol notifications described in section 5.4 notifies all subscribers of updates.

Stores that interact with the outside world, such as the filesystem store implement change logging by consulting

¹<https://www.rabbitmq.com>

```

def on_message(chan, method_frame, header_frame, body, userdata=None):
    print( userdata, body)
    chan.basic_ack(delivery_tag=method_frame.delivery_tag)

def main():
    parameters = pika.ConnectionParameters('localhost')
    connection = pika.BlockingConnection(parameters)

    channel = connection.channel() """configure channel"""
    channel.queue_declare(queue='standard', auto_delete=True)
    on_message_callback = functools.partial(
        on_message, userdata='on_message_userdata')
    channel.basic_consume('standard', on_message_callback)

    channel.start_consuming()

```

Listing 37. Simplified Python RabbitMQ receiver

external services, for example kqueue for file system operations. Stores have a method that returns their store-specific logging-store.

5.6 Dataflow Constraints

Dataflow constraints describe computed variables that are updated automatically when any of the parts of the expression, any inputs of the computation change. They are used in many different domains such as spreadsheets, build systems (make) and GUI update management to keep different objects, files or values consistent with a computed specification. They are also known as spreadsheet constraints.

As discussed in section 2.2.3, dataflow constraints are specified in Objective-S using the `|=` syntax. Listing 40 shows a simple constraint that keeps a Fahrenheit temperature consistent with a celsius temperature.

$$f \mid= c * 9/5 + 32.$$
Listing 40. Celsius to Fahrenheit constraint

Constraints use the generic logging mechanism described in section 5.5. A dataflow constraints listens to the updates that the logging store sends, and whenever one of those updates affects the variables of its right hand side, it updates the value of the variable of the left hand side by recomputing the right hand side.

With the building blocks of stores, logs and streams in place, the implementation of the generic dataflow constraint mechanism itself is only a few lines of code, a simple do-it-yourself exercise with the language only providing a small amount of syntax support.

The same building blocks should also be usable to build more sophisticated mechanisms: dataflow constraints that build an update-graph to avoid glitches and cycles, multi-way constraints as well as constraint solvers that actively search for solutions.

The fact that all the underlying mechanism are polymorphic means constraints can be applied to any kind of store. The same mechanism that is used for GUI updates can also be used for keeping directories synchronized. Listing 41 shows a script that synchronizes two directories provided as arguments on the command line.

```

#!env stsh
#-sync:<ref>sourceRef to:<ref>targetRef
source := sourceRef asScheme.
target := targetRef asScheme.
target |= source.

```

Listing 41. Directory synchronization

The Objective-S interpreter `st` converts the two command line arguments `sourceRef` and `targetRef` to references, then the script converts those references to stores. The actual synchronization is performed by the last line, which will continuously monitor the source directory and mirror any changes to the target, just like it did for the in-memory constraint.

The dataflow constraints mechanism scales beyond a single machine. Listing 42 shows a local directory being synchronized to a specified directory on a remote machine.

```

#!env stsh
#-sync:<ref>source to:host user:user dir:dir
sourceScheme := source asScheme.
session := #SSHConnection{ host: host, user: user }.
scheme:sftp := session store.
targetScheme := ref:sftp:{dir} asScheme.
targetScheme |= sourceScheme.

```

Listing 42. Remote directory synchronization

The main difference between the remote case and the local case in listing 41 is the use of an `SSHConnection` to obtain a store representing the remote filesystem. The target store

is made relative to that and the final line of code specifying the actual dataflow constraint is once again identical.

5.7 Model View Controller

The Model View Controller (MVC) architectural style [14] can be succinctly expressed using first class references and notification protocols. UI widgets are parameterized with a polymorphic reference (section 2.1.1) to the underlying model object and adopt the implicit invocation protocol (section 5.5) for model updates.

When the user changes the UI, the widget updates the model via the reference.

Model updates are logged via the observer mechanism described in section 5.5: the actual underlying model is overlaid with a Log storage combinator, the URI is logged and observers notified.

Listing 43 shows the 7 GUIs *Counter* task [13] implemented in Objective-S.

```
init:model:count := 1.
button := #Button{ title: 'count' }.
button → { model:count := model:count + 1. }.

#Grid{ rows: [
  [ #Label{ ref: ref:model:count }, button ]
] }.
```

Listing 43. Counter

Note the use of object literals to define the objects, using polymorphic references to connect the UI with a value, and using a stream connection to a closure to implement an action.

6 Applications

Both Objective-S the language and the underlying toolkits are in use in a number of different contexts.

6.1 Web Programming

As the in-process REST architectural style and its language support in Objective-S are taken from web programming, Objective-S allows direct interaction with the web without first mapping the REST style to procedures or methods.

The beneficial impact of removing these unneeded translation layers was already reported [30]. However, without language support, that solution is highly non-obvious both when writing the code and in particular when reading it. With language support, these issues fall away, as we already saw in a small example in listing 30.

Serving data on the web is also simplified, without having to resort to extensive metaprogramming or extra-linguistic annotations used in current web frameworks such as Ruby on Rails, Sinatra, Spring to map the REST architecture to procedures.

Listing 44 shows one way to say Hello World to the web using Objective-S:

```
#HTTPServer{ port: 8081 } → #MPWDictStore{ hello: '
World' }.
```

Listing 44. Hello World via HTTP

For the hello world example, a simple dictionary store is sufficient, but any store can be exposed via HTTP using the same mechanism. For example, the `env: scheme` is served using the following code: `#HTTPServer{ port: 8081 } → scheme:env`. It can then be browsed as shown in figure 3

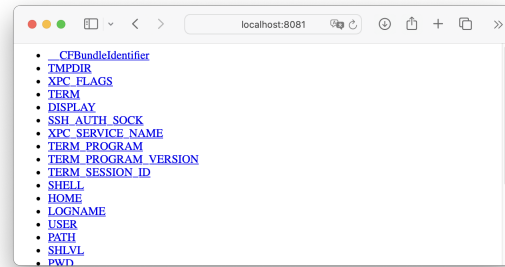


Figure 3. Environment served via HTTP

Clicking on the name of the environment variable links to the value of the variable.

This approach follows the basic principle of keeping the model of an application free from references to the outside world, while at the same time making adapters to the outside world as transparent and functionality-free as possible. This principle has been rediscovered many times, for example in the work on packaging mismatch [8], hexagonal architecture [5], MVC and naked objects.

Turning the *Todo* store from listing 3 into a web application is similarly straightforward.

Listing 45 shows a web API serving the tasks as JSON.

```
toJSON := #JSONConverter{ up: true, converterClass:
  class:Task }.
#HTTPServer{ port: 8081 } → toJson → scheme:todo.
```

Listing 45. Serving via HTTP

It uses the same JSON converter store from listing 31, but this time configured to convert to JSON when reading and from JSON when writing instead of the other way around.

Listing 46 shows a shell session interacting with this service via the curl command line tools.

```
]curl localhost:8082/tasks
[{"id":0,"done":0,"title":"Submit paper"},{"id":1,"
  done":1,"title":"Write code"}]
]curl localhost:8082/task/0
{"id":0,"done":0,"title":"Submit paper"}
]curl -X PUT localhost:8082/complete/0
]curl localhost:8082/tasks/
```

```
[{"id":0,"done":1,"title":"Submit paper"}, {"id":1,"done":1,"title":"Write code"}]
```

Listing 46. Interacting with the To Do HTTP service

We first get the list of tasks. The first task is not done yet, the second is done. We then list the task that isn't done yet, mark it as done and show the entire list again to verify.

With the string templating built into Objective-S, adding HTML support is straightforward. The Objective-S website is both built and served with Objective-S. The screenshot in Figure 4 shows the Objective-S website code and preview in *SiteBuilder*, a live web-coding tool built for Objective-S.

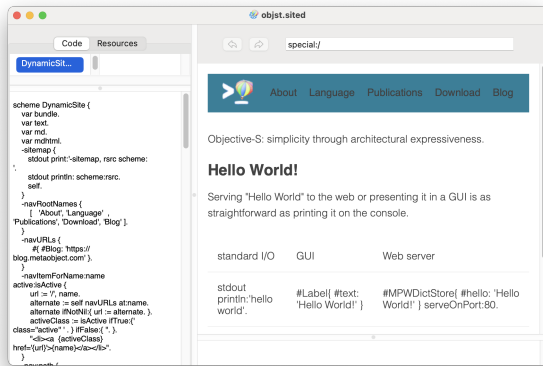


Figure 4. SiteBuilder with Objective-S site

Although *SiteBuilder* is primarily intended for the live development of dynamic web applications, it is also convenient for static sites.

6.2 GUI Tooling

As a language focused on connecting existing components, Objective-S does not come with its own GUI library. Instead it connects to the native libraries of the host system and makes it easier to define and compose UI elements and to connect them to applications.

User interface development was the original domain for which architectural-linguistic mismatch was identified, even though it was not called that at the time [4].

The problem comes in two flavors: first, defining the objects that make a UI is not inherently procedural or functional (though fluid interfaces can be used to ease construction). Second, the interaction between the UI and the model do not follow a linear control flow, but are instead characterized by complex interactions.

6.2.1 UI Definition. In Objective-S, the definition part is handled via complex object literals, which are often able to express even complex objects directly. Object templates make it possible to reuse common configurations of UI objects. Figure 5 shows the temperature converter example from the

7 GUIs suite of sample problems in a live coding environment for user interface creation called *ViewBuilder*.

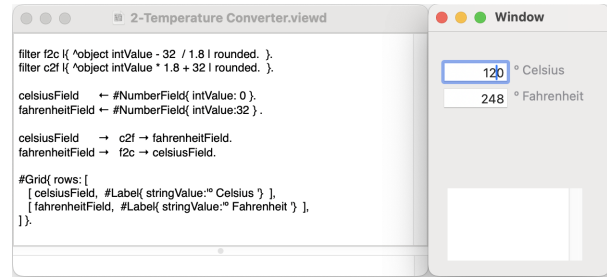


Figure 5. ViewBuilder with temperature converter

6.3 Rendering Pipeline

With Polymorphic Write Streams, Objective-S provides a built-in dataflow system. However, it is not limited to this one kind of dataflow system. Adapters can integrate other dataflow systems, and this has been done for Unix filters, the event processing systems from section 5.4 as well as the Nile dataflow language and Gezira graphics system written in Nile [11].

Figure 6 shows a screenshot of an animation of star shapes that was adapted from the Gezira project. One of the filters was replaced with a PWS filter and the mixed pipeline itself was specified in Objective-S instead of in Nile.



Figure 6. Gezira Stars Demo with mixed pipeline

7 Discussion and Evaluation

The work on Objective-S has led to many surprising discoveries. The role of connectors turned out to be both more subtle and much simpler than initially expected.

Performance on practical systems turned out much better than expected.

7.1 Role of Connectors

When the work on Objective-S started, the expectation was that connectors would be a prominent first-class language feature, as envisioned by Mary Shaw [18]. In fact, Objective-S actually still has syntax for defining connectors, but this syntax is essentially unused and not presented here. Instead, the metaobject-based implementation that was seen as a bootstrapping mechanism turned out to have exactly the right characteristics in practice.

It turns out that the number of different connectors found in the wild is small, much smaller than the number of components. The number of distinct connector types is smaller yet. Table 1 has them in the single digits, though of course larger than the number of connectors supported in mainstream programming languages.

On the other hand, these few connectors are used extremely frequently, and thus both deserve and get direct programming language support in order to make programming usable.

The polymorphism of the metaobject protocol approach allows us to bring language support to a much wider variety of connectors while keeping the set of visible language features small, by only requiring such support for the much smaller set of connector types. This makes optimized language support feasible.

7.2 Composability

Connecting components into systems requires a set of components with compatible interfaces. The same polymorphism (few connector types, more connectors) that made syntactic support feasible also lead to a relatively small set of reference interfaces that components can be made compatible with and thus a good basis for composability.

This small set of compatible interfaces enables the generic connection designator (“→”). The fact that the interfaces reflect common operations of programming languages gives them a good chance of being widely applicable.

This theoretically useful combination of a small set of interfaces with wide applicability seems to be borne out in practice: systems appear to be organized along similar lines (data access, messaging, events, data flow,...) at different scales.

This allows components to be connected without the need for glue code.

7.2.1 Glueless Connection. In UI programming, there is a conflict between the desire for glueless connection, simply hooking up UI elements to the data those elements represent, and the inevitable need for adaptation when the model does not quite match what is needed.

When glueless mechanisms such as bindings or 4GLs work, they are incredibly concise and expressive, as they match UI state directly with underlying data(base) state. Practically,

however, they only work when the underlying data model matches the UI precisely, which is rarely the case.

When it’s not the case, abstractions have to be introduced to adapt the two components to each other, and in our general purpose programming languages, these abstractions have to be procedural. Now the state-oriented UI and the state-oriented model have to interact through a mismatched procedural layer.

The result of this is that general purpose UI toolkits such as Cocoa/CocoaTouch or Android have a largely procedural and delegate-based interaction model. Not only does this mean glue code has to be written for all interactions, even the ones that go directly to data, but that glue code also does not follow the natural control-flow for procedural code, but instead requires patterns such as Inversion of Control in order for the UI components to call back to their clients.

Objective-S has facilities both for expressing dataflow and data access connections, and for abstracting over dataflow and data access. This means not only that UI/state connections can be expressed without resorting to procedural abstraction, but also that the model layer can be adapted to the requirements of the view layer without resorting to procedural abstractions.

The end-effect is that with Objective-S, glueless interaction mechanisms can be used without loss of generality. Listing 47 shows how browser views are created and presented in a window for the filesystem or environment variables, respectively.

```
#Browser{ #store: scheme:file } openInWindow:'Disk'.
#Browser{ #store: scheme:env } openInWindow:'Envit'.
```

Listing 47. Glueless UI

Although they are very different schemes, they can be presented in the same way. Other underlying data sources can be adapted to be presentable using a storage combinator and then be presented using the same single line of code.

7.3 Performance

The Objective-S work has so far focused almost exclusively on expressiveness rather than performance.

The primary current implementation is a tree-walking interpreter with the expected dismal performance characteristics. Despite this handicap, systems built with Objective-S often exhibit surprisingly good performance.

For example, the Objective-S website, served by server written in Objective-S and hosted on the entry-level free tier arm64-based hosting service has withstood several of the distributed denial of service attacks known as being shown on the Hacker News front page. Tests using the wrk² low-overhead http-testing tool show a capacity of around 1800 requests per second on that fairly low-capacity virtualized server.

²<https://github.com/wg/wrk>

7.3.1 Tasks Backend. The tasks backend developed in this paper was also benchmarked using wrk, and compared to a very similar backend implemented using the Sinatra Ruby³ web framework. The benchmark machine was a 2020 M1 MacBook with 16 GB of RAM. As only a rough indication was sought, benchmark conditions were not tightly controlled beyond the basics of doing multiple runs to prevent cold/warm effects and making sure there were no background processes.

For each framework the root page that lists all tasks was tested with two tasks. The results are shown in Figure 7, with Objective-S handling around 89K requests/second and Sinatra managing around 1.8K requests/second.

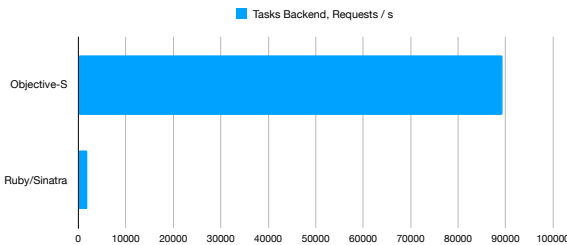


Figure 7. Tasks Backend Performance

The reason such a slow language can lead to such quick systems is that in many cases it just connects fast components written in C or Objective-C and then gets out of the way. Glueless connection and avoiding accidentally algorithmic code at the connection level is not just expressive, but also fast.

In addition, existing web programming systems often require significant parts of functionality to be expressed using their metaprogramming facilities that map REST to procedural code, which adds to overheads.

7.3.2 Native Compiler. With more Objective-S code running in the hot path, performance gets successively slower and should eventually fall below the Ruby/Sinatra example. For example, the “hello world” style server in listing 48 that is fully implemented in Objective-S only manages around 60K requests / second, despite being simpler than the tasks backend.

```

scheme HelloServer
{
  /hi/:name {
    get { 'hello: ',name. }
  }
  -main:args {
    self waitOnPort: 8081 intValue.
  }
}
    
```

Listing 48. Benchmark Hello World Service

Interacting with this server via curl yields the results shown in listing 49,

```

]curl localhost:8081/hi/Onward\!
hello: Onward!
    
```

Listing 49. Interacting with Hello World Service

Work has started on a simple native compiler based directly on Wirth’s *Grundlagen und Techniken des Compilerbaus* [31]. It does not use LLVM or similar frameworks and currently only supports ARM64 and the Mach-O object-file format as well as some JIT compilation. It does not feature any optimizations.

Benchmarking the above code with both the interpreter and the native compiler on the same machine as above yields the results given in Figure 8.

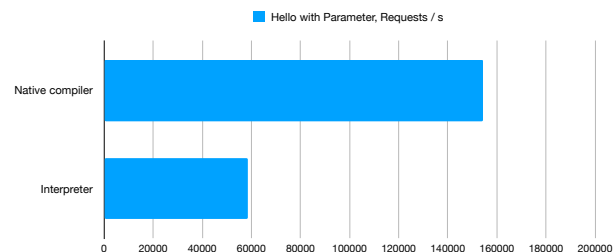


Figure 8. Native vs. Interpreted Performance

The native compiler is able to improve performance on this simple server by around a factor of 2.6.

8 Related Work

There has been a lot of work in the formalization of software architecture and coordination languages as well as in flexible, extensible programming languages. There is little if any work that combines these two approaches.

8.1 Previous Work

Some of the components that make up Objective-S and are shown here were presented in earlier work. This includes Polymorphic Identifiers [27], Storage Combinators [30], Polymorphic Write Streams [29] and Higher Order Messaging [26].

³<https://sinatrarb.com>

This work generalizes these polymorphic metaprogramming approaches into a coherent framework for organizing the metaobject protocol of a programming language along software-architectural lines, and combines them to form a programming language that is both highly extensible and capable of expressing software architecture directly in code.

Accomplishing this effect requires some additional novel elements, most notably the polymorphic connection designator “→”. Most of the heavy lifting required for polymorphic connection to work is done by the metaobject protocol, defining a few protocols that are both very composable and highly applicable.

Complex object literals help avoid “accidentally algorithmic” code for creating and configuring objects and together with the connection designator help the architecture stand out.

A number of examples are also novel, for example the use of a language-provided dataflow primitive `dataflow` to connect UI, events and observers.

The polymorphic constraints created from storage combinators and streams presented here were seen as a possibility in *Constraints as Polymorphic Connectors* [28] and declared as “future work”. At the time, the possibility seemed rather remote, but it worked out very naturally.

8.2 Architecture Description Languages

The Architecture Description Languages (ADLs) ACME [7], Rapide [17], and UniCon [20] were some of the first attempts to formalize software architecture. They allowed software architecture models to be written down and checked. Some like UniCon allowed code to be generated. Others had tools for conformance checking of implementations against the architecture.

All of them were separate from the implementation, thus not solving the architecture-implementation gap. They also tended towards verbosity, listing 50 is the ACME equivalent of the Unix shell pipe `ls | wc`.

```
System ls-wc {
  Component ls = {
    Port stdin,stdout;
  };
  Component wc = {
    Port stdin,stdout;
  }
  Connector pipe = {
    Role source,sink;
  }
  Attachments = {
    ls.stdout to pipe.source;
    wc.stdin to pipe.sink;
  }
}
```

Listing 50. `ls | wc` in ACME ADL

While ADLs clearly were on to something, making the architecture of software systems explicit, the way they did this is not useful, and they are simply not used in industry.

Instead of being more expressive due to supposed higher level of abstraction of the software-architectural view, they are actually less expressive and far more verbose. They provide no direct help implementing systems, but rather represent an additional documentation burden. Last not least, the fact that they are separate documentation that doesn’t automatically reflect the implementation, but rather has to be somehow kept in sync is antithetical to agile methods.

Objective-S makes architectural description useful. First, by taking advantage of concise syntax for connectors and reusing the same syntax with whole classes of connectors through the metaobject protocol and connector polymorphism. Second, by making the architecture be the implementation. Third, by using the architectural focus to make a wide range of architectural styles directly expressible in the code.

8.3 Smalltalk and CLOS

Objective-S is very much inspired by *The Art of Metaobject Protocol* [12], the book describing the CLOS metaobject protocol. Whereas Smalltalk’s metaobject protocol defines a point in the language design space, CLOS’s metaobject protocol describes a region, encompassing variations in various design decisions for an object-oriented language, such as how precisely method lookup works or how instance variables (slots) are stored.

By supporting multiple architectural style beyond object-orientation, the Objective-S metaobject protocol significantly enlarges the region of the total language design space covered, while at the same time still providing a well-defined conceptual model (software architecture) for navigating that space.

Other metaprogramming systems such as LISP macros or the original Smalltalk-72 provide even greater flexibility, but at the cost of not being able to provide a clear conceptual model of how to navigate the design space.

8.4 ArchJava

ArchJava [1] is an architecture-oriented extension to the Java language. It is the most direct conceptual or at least intentional predecessor to Objective-S, with the express purpose of “connecting software architecture to implementation”.

ArchJava provides linguistic support for defining, using and type-checking custom connector abstractions as an extension to the Java programming language. However, these are pure extensions and not syntactically integrated with the base language. As such, using connectors instead of the built-in mechanisms is not less but more cumbersome, and alternative architectural styles remain difficult to express directly.

Objective-S’s approach of using polymorphic connectors that are integrated via the metaobject protocol and therefore generalize rather than extend the base object-oriented language was a direct reaction to these problems of ArchJava.

8.5 CLIC

CLIC [3] is billed as a component language that is symbiotic with Smalltalk. It is the successor of several other languages that add a component languages to Smalltalk. Each successive language came closer to the concept of Objective-S, that the languages are not separate.

However, the component language always remained at least somewhat separate from the underlying Smalltalk language, being at most symbiotic. There was no integration of the component language with Smalltalk’s metaobject protocol and no support for non-call-return architectural styles.

8.6 Domain Specific Languages

Domain specific languages have been touted by many as the solution to the lack of expressiveness of general purpose languages [32].

However, DSLs and their associated tooling are costly to construct and maintain. Furthermore, they tend to have large areas of overlap with existing languages.

The Nile language referenced in section 6.3 for example duplicates variables, arithmetic operators and data structure definitions available in other languages, innovating slightly on filter methods and pipe composition.

In addition, in part because of the effort involved those facilities tend to be fairly rudimentary.

Objective-S is based on the idea that the problem with general purpose languages is not that they are too general, but actually that they are not general enough.

So instead of creating a new language that is even less expressive, the solution is to avoid all that overlap and added effort and just add the facilities required within the framework of the connectors.

For the dataflow language Nile, all the facilities required were already present and only had to be adapted to the specific filter model used in Nile/Gezira.

8.7 Plan 9

The Bell Labs Plan 9 system [16] also provides a mechanism for abstracting over and unifying storage. In contrast to Objective-S, this is abstraction is mediate by the operating system, not the language. It thus requires serialized representations, which have to be parsed, generated and accessed via the POSIX APIs. This represents significant overhead, a form of packaging mismatch.

Objective-S combines direct object interoperability with hierarchical namespaces, all without having to involve the operating system. However, Objective-S can, if a file interface is desired, export any scheme handler as a filesystem, either via the FUSE extension or a bridged Python WebDav Server.

Figure 9 shows the environment variable scheme mounted as a filesystem in the macOS Finder.

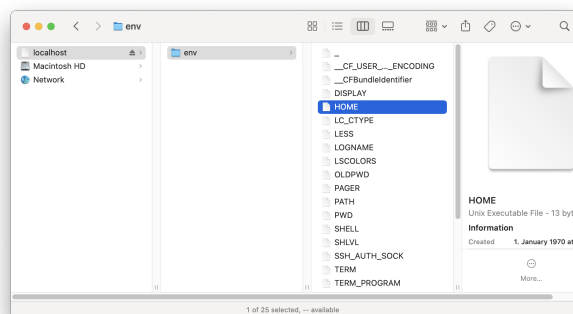


Figure 9. Unix environment scheme mounted as file system

Polymorphic Write Streams provide the in-process equivalent of the pipe facility, and are also bridged to the operating-system equivalent.

9 Summary and Outlook

In this paper we introduced Objective-S, a programming language designed to overcome architectural-linguistic mismatch and make software architecture expressible in code by means of polymorphic connectors.

The language itself provides necessary syntactic support for classes of connectors beyond procedure call, as well as the a metaobject protocol based on the notion of architectural connectors. Concrete connectors and component types plug into this framework and can then be used to natively support a variety of architectural styles.

Architectural styles supported in this way are call/return and object-orientation for procedural abstraction, in-process REST for abstraction over storage, implicit invocation, pipes and filters and last not least dataflow constraints.

Being able to both express these architectural styles directly as well as mix-and-match them as needed led to simple, straightforward and fast code in a range of problem domains that usually suffer from significant complexity and architectural mismatch.

References

- [1] Jonathan Aldrich, Craig Chambers, and David Notkin. 2002. ArchJava: connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering* (Orlando, Florida) (*ICSE '02*). Association for Computing Machinery, New York, NY, USA, 187–197. <https://doi.org/10.1145/581339.581365>
- [2] 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.
- [3] Noury Bouraqadi and Luc Fabresse. 2009. CLIC: a component model symbiotic with Smalltalk. In *Proceedings of the International Workshop on Smalltalk Technologies* (Brest, France) (*IWST '09*). Association for

- Computing Machinery, New York, NY, USA, 114–119. <https://doi.org/10.1145/1735935.1735955>
- [4] 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
- [5] Alistair Cockburn. 2005. Hexagonal Architecture. <https://alistair.cockburn.us/hexagonal-architecture/>
- [6] Brad J Cox. 1986. *Object oriented programming: an evolutionary approach*. Addison-Wesley Longman Publishing Co., Inc., USA.
- [7] David Wilkie David Garlan, Robert Monroe. 1997. *ACME: An Architecture Description Interchange Language*. Technical Report. Carnegie Mellon University.
- [8] R. DeLine. 2001. Avoiding packaging mismatch with flexible packaging. *IEEE Transactions on Software Engineering* 27, 2 (2001), 124–143. <https://doi.org/10.1109/32.908958>
- [9] Roy Thomas Fielding. 2000. *Architectural styles and the design of network-based software architectures*. University of California, Irvine.
- [10] Adele Goldberg and Dave Robson. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison Wesley.
- [11] Ted Kaehler, Bert Freudenberg, Aran Lunzer, Alan Kay, Ian Piumarta, Takashi Yamamiya, Alan Borning, Hesam Samimi, Bret Victor, and Kim Rose. [n. d.]. STEPS Toward the Reinvention of Programming, 2012 Final Report Submitted to the National Science Foundation (NSF) October 2012. ([n. d.]).
- [12] G. Kiczales, J.D. Rivieres, and D.G. Bobrow. 1991. *The Art of the Metaobject Protocol*. MIT Press. <https://books.google.de/books?id=GuOMEAAAQBAJ>
- [13] Eugen Kiss. 2018. <https://hackernoon.com/towards-a-better-gui-programming-benchmark-397aca3542b8>. <https://hackernoon.com/towards-a-better-gui-programming-benchmark-397aca3542b8>
- [14] 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>
- [15] Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. 2000. Towards a taxonomy of software connectors. In *Proceedings of the 22nd International Conference on Software Engineering* (Limerick, Ireland) (ICSE '00). Association for Computing Machinery, New York, NY, USA, 178–187. <https://doi.org/10.1145/337180.337201>
- [16] 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>
- [17] Rapide Design Team. 1997. *Guide to the Rapide 1.0 Language Reference Manuals*. Technical Report. Stanford University.
- [18] Mary Shaw. 1994. *Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status*. Technical Report CMU/SEI-94-TR-002. <https://doi.org/10.1184/R1/6582254.v1> Accessed: 2024-Apr-15.
- [19] Mary Shaw and Paul C. Clements. 1997. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In *Proceedings of the 21st International Computer Software and Applications Conference (COMPSAC '97)*. IEEE Computer Society, Washington, DC, USA, 6–13. <http://dl.acm.org/citation.cfm?id=645979.676005>
- [20] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. 1995. Abstractions for Software Architecture and Tools to Support Them. *IEEE Trans. Softw. Eng.* 21, 4 (apr 1995), 314–335. <https://doi.org/10.1109/32.385970>
- [21] Mary Shaw and David Garlan. 1996. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall.
- [22] David A Smith, A Raab, DP Reed, and AC Kay. 2006. Croquet Programming. *A Concise Guide.-Draft 0.14 [Manual de software informático]*. ViewPoints Research Institute.-Qwaq Inc (2006).
- [23] Christopher Strachey. 2000. Fundamental Concepts in Programming Languages. *Higher-Order and Symbolic Computation* 13, 1 (2000), 11–49. <https://doi.org/10.1023/A:101000313106>
- [24] Marcel Weiher. 2020. Can programmers escape the gentle tyranny of call/return?. In *Companion Proceedings of the 4th International Conference on Art, Science, and Engineering of Programming* (Porto, Portugal) (*Programming '20*). Association for Computing Machinery, New York, NY, USA, 163–172. <https://doi.org/10.1145/3397537.3397546>
- [25] 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
- [26] Marcel Weiher and Stéphane Ducasse. 2005. Higher order messaging. In *Proceedings of the 2005 Symposium on Dynamic Languages* (San Diego, California) (*DLS '05*). Association for Computing Machinery, New York, NY, USA, 23–34. <https://doi.org/10.1145/1146841.1146844>
- [27] Marcel Weiher and Robert Hirschfeld. 2013. Polymorphic Identifiers: Uniform Resource Access in Objective-Smalltalk. In *Proceedings of the 9th Symposium on Dynamic Languages* (Indianapolis, Indiana, USA) (*DLS '13*). ACM, New York, NY, USA, 61–72. <https://doi.org/10.1145/2508168.2508169>
- [28] Marcel Weiher and Robert Hirschfeld. 2016. Constraints as polymorphic connectors. In *Proceedings of the 15th International Conference on Modularity* (Málaga, Spain) (*MODULARITY 2016*). Association for Computing Machinery, New York, NY, USA, 134–145. <https://doi.org/10.1145/2889443.2889456>
- [29] 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* (Athens, Greece) (*DLS 2019*). Association for Computing Machinery, New York, NY, USA, 104–116. <https://doi.org/10.1145/3359619.3359748>
- [30] 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* (Athens, Greece) (*Onward! 2019*). Association for Computing Machinery, New York, NY, USA, 111–127. <https://doi.org/10.1145/3359591.3359729>
- [31] Niklaus Wirth. 1996. *Grundlagen und Techniken des Compilerbaus*. Addison Wesley Longman, Singapore, Singapore.
- [32] Ted Kaehler Bert Freudenberg Aran Lunzer Alan Kay Ian Piumarta Takashi Yamamiya Alan Borning Hesam Samimi Bret Victor Kim Rose Yoshiki Ohshima, Dan Amelang. 2012. STEPS Toward the Reinvention of Programming, 2012 Final Report Submitted to the National Science Foundation (NSF) October 2012. Technical Report. Viewpoints Research Institute.

Received 2024-04-25; accepted 2024-08-08