# Standard Object Out

Streaming Objects with Polymorphic Write Streams

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

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

## Abstract

We propose *standard object out*, an object-oriented analog to the output part of the Unix standard input output library.

*Polymorphic Write Streams* (PWS) act as architectural adapters between the object-oriented architectural style and the pipes and filters architectural style in the same way that stdio acts as an architectural adapter between the call/return architectural style and the pipes and filters architectural style.

Current object-oriented interfaces to the Unix I/O system mimic their procedural counterparts so closely that they manage to be neither polymorphic nor streaming, at least not for objects. Specifically the object is first converted to a fixed byte-representation by sending it a specific message and the result is then output on the underlying byte stream.

With this approach, these APIs do not allow for streaming behaviour: the entire result has to be constructed in-memory before it can be output. In addition, output of nested structures can require large multiples of time and space compared to the final output size, and fails completely if there are cycles in the object graph. It also does not allow for polymorphic behaviour.

To solve these problems, we propose *Polymorphic Write-Streams* (PWS). PWSs represent a hierarchy of classes that decouple encoding from specific streaming destinations. Using *triple dispatch* they provide streaming behaviour and allow each stream to react specifically to each kind of object and vice versa: sharing of common functionality is enabled by chaining messages along the streams' inheritance chain.

***CCS Concepts*** • **Software and its engineering → Data flow architectures**; **Object oriented architectures**; *Interoperability*; *Software performance*; Software libraries and repositories.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*DLS '19, October 20, 2019, Athens, Greece*
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6996-1/19/10…$15.00
https://doi.org/10.1145/3359619.3359748

***Keywords*** Unix, stdio, pipes, filters, objects, architecture, adapter

**ACM Reference Format:**
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 '19), October 20, 2019, Athens, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3359619.3359748

## 1 Introduction

Ever since the first occurrence of `printf("Hello World\n")`, the Unix/POSIX [5] stdio library has been a ubiquitous if unremarked part of our computing infrastructure. However, it is actually a structurally very interesting piece of software: an architecture framework that adapts between the call/return architectural style of C and similar programming languages and the pipes and filters [16] architectural style [21] of the Unix shell [8].

The call/return architectural style consists of procedures performing computations and managing state that get called and then return control to their caller, usually returning a result. The caller is suspended while the callee is running, and the callee terminates when returning to the caller.

The pipes and filters architectural styles consists of a set of filters connected by pipes. The filters can run concurrently or in an interleaved fashion, so a source filter can send multiple results to its downstream filter via the pipe, or none. Whereas procedures are in a strict hierarchy, the filters are peers.

Figure 1 illustrates the difference between the two styles schematically.

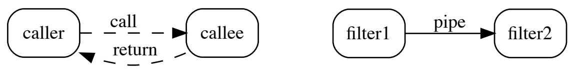

**Figure 1.** Call/return vs. pipes and filtlers architectural styles

The `stdio` library acts as an adapter between these styles. In a C program, the `printf()` [3] or `fwrite()` [1] function is accessed via call/return, meaning it gets called, performs its operation and then returns. In the context of a pipeline the program may be participating in, the `printf()` call sends data to the next step in the pipeline via the `stdout` file descriptor, and this output can be incremental, with `printf()` and other output functions called repeatedly.

The *streaming* nature means that arbitrarily large output can be generated (written to an output stream) without that data having to exist in memory at the same time. This can go as far as creating infinite output, as shown in Figure 2.

```
#include <stdio.h>
int main(void) {
   while (1) {
       printf("Infinite␣Output!\n");
   }
   return 0;
}
```

**Figure 2.** Generating infinite output from finite (and small) data

The object-oriented architectural style is a sub-style of the call/ return style [19]. Its defining characteristics are a graph of objects that can respond polymorphically to messages by invoking attached procedures called methods. Object graphs are often acyclic or even trees, but in the general case a full graph with cycles must be considered.

The standard libraries of languages supporting the object-oriented architectural style also typically provide some class or classes that interface with stdio. Their APIs closely follow the stdio example, extending it with the ability to polymorphically print objects where stdio supports only strings and primitive types.

This ability to print objects is implemented by sending the object to be printed a message that tells the object to convert itself to a string. This string is returned and then output. This approach appears perfectly natural in the object-oriented style and is the approach taken by essentially all major OO languages except C++ [20].

This staged approach of first generating a string representation as a result of a message send and then outputting that string representation results several unwanted behaviours: first, it inhibits extensibility in the kinds of output streams and representations, second it necessarily leads to potentially severe performance problems with deep graph/tree structures and third it cannot handle graphs with cycles at all, resulting in an infinite recursion and consequently stack overflow crashes.

In short, adapting between the object-oriented architectural style and the pipes and filters architectural style in this fashion results in losing the main attributes of both styles: polymorphic behaviour, the ability to deal with arbitrary object graphs, and streaming behaviour.

We propose Polymorphic Write-Streams (PWS) as an extension of the double-dispatch approach as a generic replacement for the object-oriented equivalent of the stdio library. As architectural adapters, PWSs combine the benefits of the pipes and filters architectural style with the advantages of the object-oriented architectural style.

Section 2 will demonstrate the problems of using standard stream output mechanisms with a typical object-graph. Section 3 will introduce Polymorphic Write-Streams, with Section 4 presenting alternative intermediate representations. Section 5 shows how PWSs can be used to stream results. Section 6 describes real-world applications that benefit form PWSs, with a more detailed evaluation following in Section 7. Section 8 will look at some of the implications of this work. Related work is discussed in Section 9. Section 10 provides a summary and a look at future work derived from the observations from Section 8.

## 2 Motivating Example

The Node class in Figure 3 serves as a stand-in for an object graph. It has textual content and `left` and `right` child nodes, any of which may be `nil`. The `description` method creates a textual representation of the object.

```
@import Foundation;

@interface Node:NSObject

@property (nonatomic,strong) Node *left,*right;
@property (nonatomic,strong) NSString *s;

@end

@implementation Node

-(NSString*)description
{
    return [NSString stringWithFormat:
            @"<\%@:\%p␣string:␣\%@␣left:␣\%@␣right:␣\%
                @>",
               [self class],self,self.s,self.left,self
                   .right];
}

@end
```

**Figure 3.** A simple node object in Objective-C

Objective-C uses Smalltalk-style keyword syntax mixed with C types for messages and method definitions, with the minus sign introducing instance methods and the plus sign introducing class methods.

Figure 4 shows how to create a single Node object and printing it using the `NSLog()` function that is a standard part of the Cocoa frameworks [6].

`NSLog()` is similar to the `stdio` function `printf()` in that it takes a string with format specifiers and a variable number of arguments, and outputs the result to a pre-defined Unix file descriptor. There are some differences: in addition to the `printf` format specifiers such as `%d` for integers it also accepts the format specifier `%@` for objects. This causes the

```
int main(void ) {
    Node *a=[Node new];
    NSLog(@"Node:␣\%@",a);
}
Output:
Node: <Node:0x7fc3f5605610 string: (null) left: (null
     ) right: (null)>
```

**Figure 4.** Printing a simple node, with output

object to be sent the `description` message to return a string representation. In addition, `NSLog()` outputs to `stderr` instead of `stdout` and prepends a timestamp and appends a newline.

This functionality is not unique to Objective-C, Ruby [27] has a `print` method that sends the `to_s()` message, Java [11] has `PrinStream` (usually accessed via `System.out`) and the Python [22] standard library uses `__str__()`. In what follows, we will be using Objective-C.

## 2.1 Flexibility

Since the interface between the stream and the objects is the single `description` message (or equivalent for other systems), output is limited to that specific serialisation format.

Creating a different output format, even if it is only a minor variant, requires introducing a completely different serialisation message, which must then be implemented by all objects, even if the stream defining the format is a subclass of an existing stream.

While it is possible to provide a default implementation that defers to a previous implementation, that approach will result in inappropriate representations being written, even for objects that have a correct implementation.

As an example, and taking the `Node` class of Figure 3, we could introduce a new debug output stream that sends objects written to it the `debugDescription` message instead of `description`. This `debugDescription` creates concise output for large data-sets, so for example `NSString` has an implementation that, for long strings, prints just the beginning and end of the string as well as a message stating how many characters were omitted.

If `Node` does not yet implement `debugDescription` and reverts to `description` instead, it will then also send the `description` message to its string instance variable, despite the fact that we asked for a `debugDescription` and `NSString` implements `debugDescription`.

## 2.2 Performance

One of the key features of the pipes-and-filters architectural style is that it is capable of incremental, streaming processing, meaning a filter doesn't have to hold all the data it processes in memory at once, unless this is required by the semantics of the operation.

When constructing the output is done by sending a single message and then outputting the result, it is clear that the entire output has to be present in memory at once. In this case, it is difficult to speak of streaming, the architectural style is closer to batch-sequential, even if the intermediate results are not saved to disk.

The code in Figure 5 constructs a tree of `Node` objects with a string payload of specified depth.

```
+(instancetype)tree:(int)depth string:(NSString*)s
{
   Node *n=[[self new] autorelease];
      n.s=s;
      if ( depth > 0 ) {
         n.left = [self tree:depth-1 string:s];
         n.right= [self tree:depth-1 string:s];
      }
   return n;
}
```

**Figure 5.** Constructing a balanced tree of specified depth

Figure 6 uses this tree construction method to create a balanced tree of depth 16 with a small string payload at each node.

```
int main(void ) {
    Node *root=[Node tree:16 string:@"Hello␣World"];
    NSLog(@"Node:␣%@",root);
}
```

**Figure 6.** Generating and printing a deep tree

The tree generated by this code consists of roughly 130K nodes taking 5MB of memory and producing 9.5MB of textual output. Producing that output takes over 200MB of memory, so 40 times the in-memory size and 20 times the size of the final output produced.

The reason for this expansion is that at each level of the tree, complete temporary results have to be constructed and then included in the result of the next higher level.

Knowing the cause, we can construct a degenerate case that will highlight the problem: a tree of maximum depth that has a large payload at its leaf and minimal payloads for the interior nodes.

Figure 7 constructs such a simple degenerate tree of adjustable depth with a very large string at its single leaf node.

Using this method with a `depth` of 200 and a `leafSize` of 10MB creates a tree that takes 12MB in memory and produces output of 10.4MB, but requires 1.98 GB of memory to produce, for an expansion factor of roughly 200x. This expansion factor is proportional to the depth of the tree, so a `depth` of 1000 requires almost 10GB to produce, whereas both the in-memory size of the tree and the output hardly increase in size.

```
+(instancetype)onlyLeft:(int)depth leafSize:(int)mb
{
    long string_length=mb*1024*1024;
    NSMutableString *s=[[NSMutableString alloc]
         initWithCapacity:(mb+1)*1024*1024];
    while ( [s length] < string_length ) {
        [s appendString:@"Hello␣World!"];
    }
    Node *top=[self node];
    top.s=[s autorelease];
    for (int i=0;i<depth;i++ ) {
        Node *n=[Node node];
        n.left = top;
        n.s = @"";
        top=n;
    }
    return top;
}
```

**Figure 7.** Create a degenerate graph with a large leaf node

### 2.3 Stability

As an example, Figure 8 implements a very simple graph with a cycle.

```
int main(void ) {
    Node *a=[Node new];
    Node *b=[Node new];
    a.left = b;
    b.left = a;
    NSLog(@"Node:␣%@",a);
}
Output:
Segmentation fault: 11
```

**Figure 8.** A graph with cycles cannot be serialized

Due to the cycle, attempting to output this graph necessarily results in a crash.

## 3 Polymorphic Write-Streams

Polymorphic Write-Streams (PWS) are the object-oriented equivalent of the `stdio` library's `FILE` structure and associated functions.

A PWS consists of a number of elements:

1. A low level interface for outputting bytes, equivalent to the output functions of stdio.
2. An interface for writing objects to the stream
3. An interface for double-dispatching back to the object with the name of the stream via a stream-specific *stream-writer message* so objects can react to specific streams.
4. An API for deconstructing objects provided by the stream that is used in a triple-dispatch by the object back to the stream once it has identified the specific stream kind via the double dispatch. The triple-dispatch identifies the specific kind of object and its sub-structure to the stream.
5. A way of automatically chaining stream-writer message implementations in a hierarchy of messages that reflects the inheritance hierarchy of the streams.

The whole interaction is shown in the sequence diagram of Figure 9.

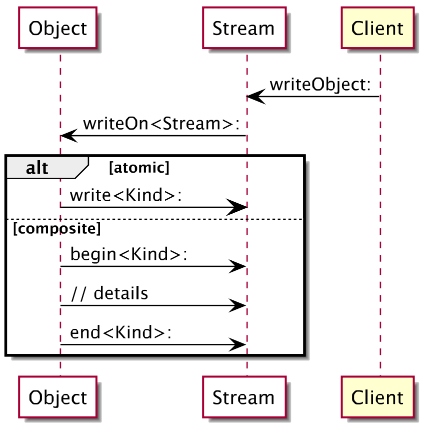

**Figure 9.** Sequence diagram for one cycle of the object/PWS interaction

PWSs have been implemented in several programming languages including Objective-C, Squeak/Smalltalk and C#. Here, we will look at the Objective-C implementation available as part of MPWFoundation [23].

Specifically, MPWFoundation includes the `MPWByteStream` class that implements the PWS concept.

### 3.1 Byte Streaming Interface

As a basis, `MPWByteStream` needs to implement an interface for dealing with outputting bytes and strings, essentially a wrapper over the underlying byte-oriented `stdio` interface. This `ByteStreaming` protocol (Figure 10) defines messages for outputting raw bytes as well as formatted strings. Some convenience message are omitted here for clarity.

```
@protocol ByteStreaming

-(void)appendBytes:(const void*)data length:(
    NSUInteger)count;
-(void)printf:(NSString*)format,...;
-(void)printLine:(NSString*)format,...;

@end
```

**Figure 10.** ByteStreaming protocol

With this protocol in place, we can now define the class MPWByteStream as implementing that protocol, as seen in Figure 11.

```
@interface MPWByteStream : NSObject<ByteStreaming>
{
   id byteTarget;
}
```

**Figure 11.** MPWByteStream PWS class

The byteTarget holds an adapter that sends the actual bytes to different targets depending on how the instance of MPWByteStream is initialized: a FILE or a low-level *fd*, an NSData object that holds raw bytes or an NSString of unicode text.

### 3.2 Object Interface

The object-oriented interface of PWSs is the Streaming protocol shown in Figure 12. It defines a single message, writeObject:, which takes a single argument.

```
@protocol Streaming

-(void)writeObject:anObject;

@end
```

**Figure 12.** Streaming Protocol

So in order to send an object to stdout, all a filter has to do is get the PWS for stdout and then write the object to that PWS, as shown in Figure 13.

```
   [[MPWByteStream Stdout] writeObject:object];
```

**Figure 13.** Write object to stdout

The stream and the object then cooperate to determine what byte-representation to write to stdout. The stream responds by dispatching back to the object with information as to the actual stream being written, and the object then responds by writing its structure to the stream.

### 3.3 Double Dispatch

The PWS responds to the writeObject: with a double dispatch [13] back to the argument object (Figure 14).

The stream sends the argument object a message that encodes the class of the stream in the name of the message, with the stream itself as an argument. The format of the message follows the pattern writeOn<StreamClass>:, so in case of the MPWByteStream above the objects gets sent the writeOnMPWByteStream: message with the MPWByteStream instance as its sole argument.

```
-(SEL)streamWriterMessage
{
   NSString *selName=[NSString stringWithFormat:@"
       writeOn%@",[self class]];
   return NSSelectorFromString( selName );
}

-(void)writeObject:anObject
{
       NSValue *p=[NSValue valueWithPointer:anObject
           ];
       if ( ![alreadySeen containsObject:p]) {
           [alreadySeen addObject:p];
           [anObject performSelector:[self
               streamWriteMessage] withObject:self];
           [alreadySeen removeObject:p];
       } else {
           [self printFormat:@"<Already␣seen:␣%@:%p>"
               ,[anObject class],anObject];
       }
}
```

**Figure 14.** Double dispatch and short circuit

In addition it adds the pointer of the object to a set of pointers it has already seen for the duration of outputting that object, and short-circuits the process if the object has been seen. This avoids infinite recursion for graphs with cycles.

### 3.4 Triple Dispatch

Once the stream has identified itself to the object, the object gets to write an appropriate representation of itself to the stream. The stream directly supports writing a number of primitive objects such as numbers and strings and also support writing composite objects such as dictionaries and arrays and the key/value pairs contained within dictionaries and objects.

Objects other than the receiver itself are again written to the stream using the writeObject: message, restarting the triple-dispatch mechanism.

```
-(void)writeOnMPWByteStream:(MPWByteStream*)s
{
   [s printFormat:@"<%@:%p␣string:␣",[self class],
       self];
   [s writeObject:self.s forKey:@"string"];
   [s writeObject:self.left forKey:@"left"];
   [s writeObject:self.right forKey:@"right"];
   [s printFormat:@">"];
}
```

**Figure 15.** Writing a Node object on a stream

### 3.5 Message Chaining

So far, every new stream, with its own stream-writer message, requires that all objects that might conceivably be serialised implement that stream-writer message (so the pattern `writeOn<Stream>:`). As noted in Section 2.1, such a requirement is onerous, even with class extensions.

To avoid implementing all stream writer messages on all objects, we chain those messages similar to the superclass dispatch in message dispatch: in the root class (`NSObject` for Objective-C), we implement a specific stream's stream-writer message by sending the super-class's stream-writer message to `self`, as shown in Figure 16.

```
@implementation NSObject(DebugStreaming)

-(void)writeDebugStream:(DebugStream*)aStream
{
    [self writeOnMPWByteStream:aStream];
}

@end
```

**Figure 16.** Message chaining

This reduces the requirement for creating a new stream from adding the new stream writer messages to all objects in the system to adding just a single method to `NSObject`, a reduction that makes the concept feasible.

However, even having the create this single extra method per PWS adds unnecessary boilerplate, especially as the method is completely mechanical. Figure 17 shows the code used to generate this method automatically when the class is first used.

```
+(void)initialize
{
    if ( ![self instancesRespondToSelector:mySelector
        ]) {
      SEL superSelector = [[self superclass]
          streamWriterMessage];
      IMP theImp=imp_implementationWithBlock( ^(id
          blockSelf, id stream ){
           (objc_msgSend)(blockSelf, superSelector ,
                stream); }
                                        );
        class_addMethod([NSObject class], [self
            streamWriterMessage], theImp, "v@:@");
    }
}
```

**Figure 17.** Generating the message chaining method automatically

The `+initialize` message is sent to every Objective-C class before it is used. The basic idea is to create the method and then add it to the `NSObject` class using the Objective-C runtime function `class_addMethod()`.

In Objective-C, a method body can be dynamically created at run time from a block using the Objective-C runtime function `imp_implementationWithBlock()`, which takes the block as its argument and returns a method pointer (`IMP`) that can be added to the runtime.

The method body is defined by the block that is the argument to `imp_implementationWithBlock`. This block directly calls the `objc_msgSend()` runtime function that is used to send messages in Objective-C.

If generating methods automatically is not possible, that chaining method can also be written by hand for every stream as shown above. A slower alternative would be for the stream to check if the object in question responds to the stream writer messages of itself and its superclasses in turn.

### 3.6 Stream Targets

PWSs do not generate output directly, but instead delegate this to a `target` by sending it the serialised bytes using the message `appendBytes:length:`. Targets exist to write those bytes to a Unix file descriptor via the `write()` system call, to a *stdio* `FILE*` via `fwrite()`, or to either an `NSString` or `NSData` object, the former being a unicode text string, the latter an uninterpreted sequence of bytes.

Another target draws the output generated directly to a GUI text view, without having to indirect through a pseudo terminal, similar to the text streams in the Common Lisp Interface Manager (CLIM) [17].

As the `appendBytes:length:` message is also part of the `ByteStreaming` protocol, there is also potential for stacking streams.

### 3.7 Thread Safety

Polymorphic Write Streams are not thread-safe. The result is an inherently serial Unix byte stream sent to `stdout`, and at least the serialisation of one top-level object has to be atomic, with no interfering writes, or the result will be garbled.

If multiple threads need to write objects to `std-object-out`, the recommended method would be for each to have its own PWS, with intermediate results buffered and written to the Unix output when the client signals completion.

## 4 Alternative Materialised Intermediate Representations

Previously, we identified the core problem of most languages' approach to `stdio` as first generating a string representation as a result of a message-send and then outputting that string representation. PWSs solve this problem by never fully materialising this string representation, but instead generating it incrementally and immediately outputting it to the output stream as it is generated.

Another approach is to generate a different intermediate representation as the result of the message send, and outputting that representation.

### 4.1 Property Lists

NeXT Property Lists [4] are both an in-memory representation and one of several file formats. The in-memory representation consists of dictionaries, arrays, as well as string, number and date objects. The file formats include ASCII property lists, XML property lists, binary property lists and JSON.

Creating a serialised representation is a two-step process: first, user code creates the in-memory representation, then library code converts that in-memory representation to a serialised representation that can then be saved to disk.

Like PWSs, this process can avoid materialising the full string representation in memory using a similar technique to serialise this specialised property list. The difference to PWSs is that the serialisation code only has to deal with a few well-known data types and can be highly optimised for incrementally generating representations of those types.

However, the in-memory property list representation has to be materialised fully, and will typically roughly match the structure of the object tree to be materialised. Being generic, this intermediate representation tends to be both significantly larger and slower to process than the original object graph, as discussed in depth in *iOS and macOS Performance Tuning: Cocoa, Cocoa Touch, Objective-C, and Swift* [26].

Another limitation is that the format is only suitable for generating file formats that correspond to property lists, not arbitrary streams of data.

### 4.2 Ropes

Another way of mitigating the issues of a materialised intermediate string representation would be to use more advanced string data structure, such as ropes [7]. Ropes represent a string not as a linear sequence of characters, but as a tree of nodes that represent the operations that would be needed to generate the sequences of characters.

This representation is advantageous for nested trees. Generating the representations for the intermediate nodes does not require allocating a full buffer for the entire intermediate character sequence and then copying all the previous character sequences, an $O(n)$ operation per level encountered. Instead, each copy operation just requires allocating the rope node for the operation, an $O(1)$ operation per level.

Since a Unix byte-stream *is* a linear sequence of characters, the rope structure has to be converted to that linear format at some point. This can occur either in a single call/return style step, with many of the same problems as before, or incrementally just like PWSs. In a sense, adding a rope doesn't solve the problem of creating the fully serialised representation from a tree, it just delays it, at the cost of a materialised temporary tree structure.

Ropes and property lists are very similar in that they define an in-memory tree of specialised types or a specialised type. If created by call/return, they both fail when the original object graph contains cycles. They both will create what is essentially a duplicate of the original object graph or tree, which then still has to be serialised.

## 5 Streaming vs. Materialisation

As we saw in the previous sections, PWSs eliminate the need for a fully materialised intermediate representation of the object graph, be that as a linear string, a property list or a rope, a characteristic shared with many object serialisation mechanisms.

Unlike those serialisation mechanisms, PWSs can also eliminate the need for a fully materialised version of the original object graph itself, or for any materialised version at all.

The code in Figure 18 adjusts the tree-construction code from Figure 5 to not actually construct the tree, but instead just output what the constructed tree would have output to `stdout`.

```
+(void)printTree:(int)depth string:(NSString*)s
       onStream:(MPWByteStream*)stream
{
   [stream beginObject:self];
   [stream writeObject:s forKey:@"string"];
   if ( depth > 0 ) {
   [stream writeKey:@"left"];
       [self printTree:depth-1 string:s onStream:
           stream];
   [stream writeKey:@"right"];
       [self printTree:depth-1 string:s onStream:
           stream];
   }
   [stream endObject:self];
}
```

**Figure 18.** Writing a non-materialised tree

Not needing a fully materialised object graph in memory opens up the possibility of streaming object applications, if the streaming-friendly PWSs are matched with comparable technology on the input side or an infinite stream generator similar to the code in Figure 2.

Although streaming of non-materialised or at least not fully materialised is the norm for Unix filters, and basic access to the Unix I/O functions is present in OO libraries, developers usually face a hard dichotomy: either stream at the level of individual strings, without OO abstractions, or use object abstractions and give up streaming.

With PWSs, objects can be combined with streaming at any level of granularity desired.

## 6 Applications

PWSs have been used in diverse application ares ranging from classic object serialisation tasks to PDF and Postscript generation. The performance characteristics of PWSs mean

### 6.1 Printing and Logging

The application shown in the examples thus far is one of the most common: simple printing and logging of both unstructured and structured data to `stdout` or `stderrr`. The benign behaviour of PWSs with regards to deeply nested graphs or cycles means that objects can be logged safely without having to worry about unexpected errors or performance degradation.

Having a class hierarchy of PWSs rather than a single function means that variations can be accommodated, for example one PWS will only print the start and end of a large string, while another will output the entire string. Deep graphs can be abbreviated by only printing them to a specific depth.

PWSs can also change the work distribution between filters or commands and the output system of an interactive, object-oriented shell [25]. Instead of providing various formatting options for the `ls` command the way it is done in Unix, the equivalent directory listing command just outputs directory entry objects to the current PWS. Different PWSs format the output accordingly.

### 6.2 Object Serialisation

Object serialisation is an obvious application of PWSs, and the MPWFoundation library [23] includes stream subclasses for generating JSON as well as the NeXT/Apple property lists formats presented in Section 4.1.

The different PWSs share much of their common functionality and perform significantly better than non-PWS APIs provided by macOS and iOS, both in execution time and particularly in memory use [26].

### 6.3 Postscript

Pre-press applications were one of the incubators for the PWS idea: at the time, the files that had to be dealt with often outstripped the memory of the machines that needed to process those files. It was typical to have to generate a 90MB Postscript file and then rasterise that to a 128MB bitmap for output on a color laser copier, all on a machine with no more than 32MB of main memory.

Memory-frugal streaming approaches were thus not just a nice-to-have optimisation, but absolutely essential for programs to process the amount of data needed on the machines available.

Figure 19 shows part of the definition of a stream that's specialised for generating Postscript code [18] for use in digital typesetters and printers..

```
-initWithTarget:newTarget
{
   self = [super initWithTarget:newTarget];
   lineto = @"%g␣%g␣lineto\n";
   moveto = @"%g␣%g␣moveto\n";
   curveto = @"%g␣%g␣%g␣%g␣%g␣%g␣curveto\n";
   closepath = @"closepath\n";
   linewidth = @"%g␣setlinwidth\n";
   clipExists = NO;
   actions=[self actions];
   return self;
}
-(SEL)streamWriterMessage
{
   return @selector(writeOnPSByteStream:);
}
-(void)writeArray:(NSArray*)anArray
{
   [@"[␣" writeOnPSByteStream:self];
   [super writeArray:anArray];
   [@"␣]␣" writeOnPSByteStream:self];
}
-(void)writeDictionary:(NSDictionary*)dict
{
   [@"<<␣" writeOnPSByteStream:self];
   [super writeDictionary:dict];
   [@"␣>>␣" writeOnPSByteStream:self];
}
-(void)writeObject:anObject forKey:aKey
{
   [@"/" writeOnPSByteStream:self ];
   [[aKey stringValue] writeOnPSByteStream:self];
   [@"␣" writeOnPSByteStream:self];
   [anObject writeOnPSByteStream:self];
   [@"␣" writeOnPSByteStream:self];
}
-(void)lineto:(float*)coords
{
   [self printf:lineto,coords[0],coords[1]];
}
-(void)moveto:(float*)coords
{
   [self printf:moveto,coords[0],coords[1]];
}
-(void)writePath:(MPWPath*)aPath
{
   [aPath pathForall:self];
}
...
```

**Figure 19.** Postscript write stream

In addition to the initialiser and the explicit definition of the stream-writer message, the code defines three basic kinds of methods: first, specialisations of the more general methods for writing arrays and dictionaries, which generate the syntax required by Postscript for these elements. Second, Postscript-specific primitive methods such as `lineto:` and `moveto:` and finally a Postscript-specific higher level method, `writePath:`, which is used by path objects when they are sent the `writeOnPSByteStream:` message.

The `pathForall:` method on paths writes the individual path elements to the stream using methods such as as `moveto:` and `lineto:`.

The graphical objects and the Postscript stream define a private sub-protocol that allows them to optimally generate code. For example, a specialised path subclass was created to store its paths in a format directly compatible with the Display Postscript (DPS) binary object format. When writing such a path to a specialised DPS output stream, the path and the stream would collaborate to directly send that stored binary representation without any further encoding, while at the same time writing the ASCII representation on other kinds of Postscript output streams.

### 6.4 PDF

Where Postscript is a flat file format, a single textual stream of instructions to be executed by the output device, the successor Portable Document Format (PDF) [2] is a structured file format: the top-level file consists of a series of structured objects, which are indexed for random access. These structured objects resemble Postscript dictionaries, with some having a data stream attached to them.

Some of these data streams in turn contain drawing commands that are equivalent to a stylised version of Postscript commands, without the programmability. Streams can be compressed.

Generating drawing commands into a compressed data stream that is compressed and in turn written incrementally into the top-level PDF file is handled by stacking several PWSs using the target facility described in Section 3.6.

The textual drawing commands are generated by a PWS that is a subclass of the Postscript generator of the previous section. The output of that PWS is passed to a streaming flate compressor, a simplified PWS that compressed the data sent to it via the `appendBytes:length:` and writes the compressed output to its target, again via `appendBytes:length:`. The final target is another PWS that writes dictionaries and streams, keeping track of the exact byte position of each dictionary and stream written to add to the PDF index that is written at the end of the file.

### 6.5 XML and HTML

XML and HTML PWSs use the multiple-dispatch mechanism to introduce their own private message protocol for creating XML/HTML content while at the same time remaining compatible with other, more general output streams.

The Objective-Smalltalk website is served using a PWS that generates HTML [24].

## 7 Evaluation

In Section 2, we noted several problems, including lack of flexibility, performance and crashes on cyclic structures. The previous section already showed how flexibility is improved via triple-dispatch and chaining.

The crash on cyclic structures is avoided by the PWS refusing to follow cycles. In order to do that, it keeps track of the objects that are currently in the process of being output. If it encounters such an object again, it short-circuits the process, as seen in Figure 14.

### 7.1 Performance

We already noted some performance figures for producing string representations in Section 2. In this section, we look at performance in greater detail and demonstrate that PWSs solve the performance issues encountered earlier.

All performance tests were run on a MacBook Pro (13-inch 2018) with a four core 2,7 GHz Intel Core i7 processor configured for 8 hyperthreads, with 16GB of 2133 MHz LPDDR3 physical memory. Time was measured using the `time` command line program, memory consumption was measured using the `mstats()` library call. Output was directed to a file on the locally attached SSD and size measured after the run using the `wc` command.

Figure 20 shows the code for creating the degenerate graph using the `description` method used implicitly by the `NSLog()` function.

```
int main(int argc, char *argv[]) {
   Node *root=[Node onlyLeft:atoi(argv[0] leafSize
       :10];
   [[MPWByteStream Stdout] writeObject:root];
}
```

**Figure 20.** Output degenerate graph via NSLog() and description

Figure 21 plots the memory and CPU consumption of runs of the code in Figure 20 with graph depths ranging from 0 to 1000.

Although both the in-memory size of the tree and the total output size stay essentially constant at 12MB, both time to process and memory used grow linearly with depth, with memory consumption rising to 11 GB for a graph with depth 1000.

Figure 22 shows the identical graph being constructed, except this time the output is via the `MPWByteStream` Polymorphic Write-Stream.

Figure 23 plots the memory and CPU consumption of the PWS.

Note that although the depth axis is identical between Figures 21 and 23, the y-axis had to be rescaled in order to reasonably fit the same information: CPU consumption is now shown in 10s of milliseconds instead of seconds, and

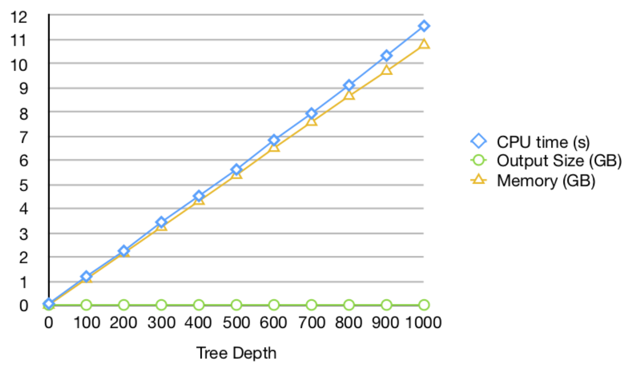

**Figure 21.** Memory and CPU use for producing a string representation via NSLog()

```
int main(int argc, char *argv[]) {
    Node *root=[Node onlyLeft:atoi(argv[0] leafSize
        :10];
    [[MPWByteStream Stdout] writeObject:root];
}
```

**Figure 22.** Output degenerate graph via PWS

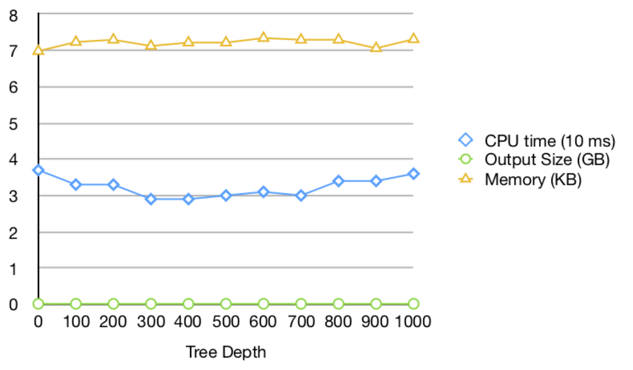

**Figure 23.** Memory and CPU use for producing a string representation via PWS

temporary memory consumption is shown in Kilobytes instead of Gigabytes.

For the tree of depth 1000, the PWS version uses slightly more than one million times less memory and uses 30 times less CPU time. Memory consumption is not just constant regardless of depth, which is the desired outcome for a streaming operation, but also less than the output size.

The fact that memory consumption remained constant regardless of depth may have been an artifact of the construction of the degenerate tree, which also had nearly constant output size and in-memory size. Figure 24 uses the balanced tree constructor to create and then output trees of increasing sizes.

```
int main(void ) {
    Node *root=[Node tree:16 string:@"Hello␣World"];
    [[MPWByteStream Stdout] writeObject:root];
}
```

**Figure 24.** Printing a tree using a PWS

As can be seen from the results graphed in Figure 25, output size now increases, but memory consumption is still constant.

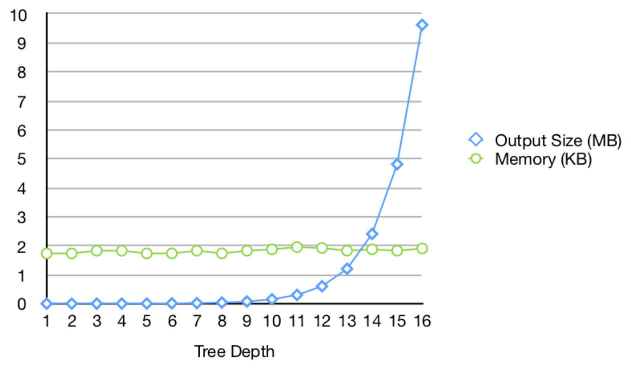

**Figure 25.** Memory use for producing a string representation via PWS

## 7.2 Dispatch Overhead

Where the previous section compared PWS to traditional OO approaches, this section will compare PWS to tradition streaming approaches.

Figure 26 reproduces the balanced-tree scenario from Figures 3, 5, and 24 in plain C, without any message dispatch.

Both this C code and the code for the comparable PWS were run with output directed to a file. The file output was compared with the Unix `diff` command to ensure that both programs produce the same output.

Figure 27 plots the wall-clock times for node tree depths from 16 to 23: except for very small sizes, the PWS actually performed better than the C version.

Some but not all the performance difference is due to the `stdio` library's buffer size, which is too small for current I/O hardware and causes significant system call overhead. Figure 28 shows that removing system call overhead reduces the gap without eliminating it.

Increasing the buffer size for the C version to 10MB using the `setvbuf()` library function further reduces the gap, still without entirely eliminating it.

Although these results do not clearly show what the overhead of triple dispatch is, they do show that is small enough

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct Node {
   char *s;
   struct Node *left;
   struct Node *right;
} Node;

void printNode( Node *n, FILE *out ) {
   if (n) {
      fprintf(out, "<Node:string:␣%s;\n",n->s);
      fprintf(out, "left:␣");
      printNode(n->left,out);
      fprintf(out, ";\n");
      fprintf(out, "right:␣");
      printNode(n->right,out);
      fprintf(out, ";\n");
      fprintf(out,">");
   } else {
      fprintf(out, "(null)");
   }
}

Node *tree( int depth, char *s ) {
   Node *n=calloc( 1, sizeof(Node));
   n->s = s;
   if ( depth > 0 ) {
      n->left = tree( depth-1, s );
      n->right= tree( depth-1, s );
   }
   return n;
}

int main(int argc, char *argv[]) {
    int depth=argc > 1 ? atoi(argv[1]) : 10;
    Node *root=tree( depth, "Hello␣World!");
    printNode( root, stdout );
}
```

**Figure 26.** C version of tree generation and output

to be easily overwhelmed by other factors such as appropriate sizing of I/O buffers.

### 7.3 Threats to Validity

On potential issue is that the problems encountered were specific to Objective-C and its Foundation library. In order to ensure that this is not the case, we reproduced the problematic results with a number of other languages.

The results for Ruby are shown in Figure 29.

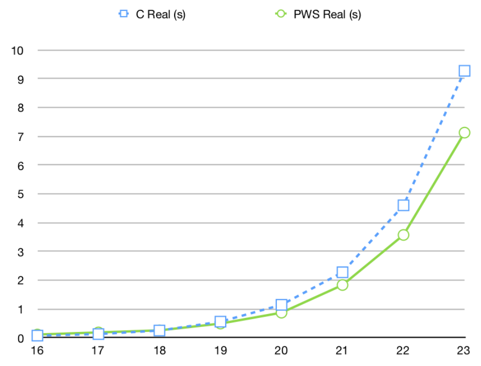

**Figure 27.** C-Tree vs. PWS performance: total time

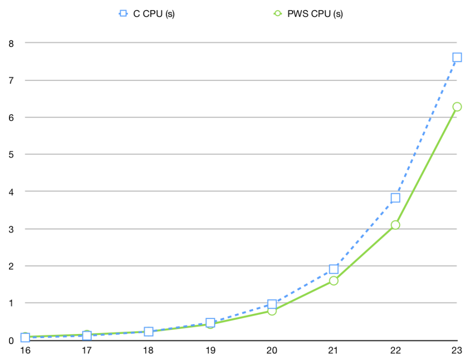

**Figure 28.** C-Tree vs. PWS performance: CPU time

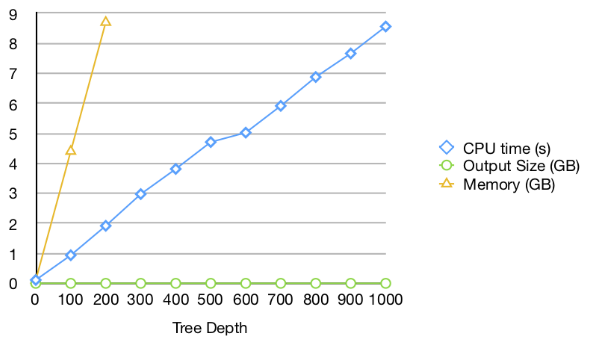

**Figure 29.** Ruby results

The results show the same general behaviour as observed for Objective-C, with slightly less CPU usage and significantly higher memory consumption. In fact, memory consumption was sufficiently high that capturing that information was stopped for higher tree depths.

Alternative intermediate representations were discussed in Section 4 but not separately measured.

## 8 Discussion

PWS solves the problems encountered by the standard architectural adapters between the object-oriented architectural style and the Unix pipes and filters architectural style: it is flexible, can protect itself against cycles in the object graph and has the performance characteristics we would expect of a filter.

The downside is a slightly higher initial implementation effort, but as this really is small and only required once, a PWS-like mechanism could be the standard way of adapting object-orientated styles to pipes and filters, just as `stdio` is the standard adapter for general call/return architectural styles.

One non-obvious aspect of the way PWS solves the problems created by relying on return values for encoding is that those problems are structural. They are not due to deficits in the implementation. Once it is decided that the API will consist of sending a message to an object with the result in the return value, the problem is not addressable.

The problem is only fixable by changing the API contract and passing an accumulator for the result into the messages, so that the result can be constructed incrementally.

Going in the other direction, on the other hand, is trivial: if we find the PWS interface to be too cumbersome for just returning a string result, a simple wrapper (Figure 30) will suffice.

```
-(NSString*)description
{
   MPWByteStream *s=[MPWByteStream stream];
   [s writeObject:self];
   return [s target];
}
```

**Figure 30.** Wrapper adapting a PWS to call/return style

We can implement a `description` method by creating an instance of `MPWByteStream` with a string target, writing the receiver to that stream and returning the accumulated string from the `MPWByteStream`'s target.

This asymmetry between the call/return style and a more streaming-oriented approach suggests that streaming is the more fundamental style. That is: we can easily obtain a call/return implementation given the streaming implementation, but not the other way around. This in turns suggests that implementations of algorithms dealing with object graphs should, if possible, aim for a streaming implementation first, even if that appears unnatural in the object-oriented architectural style.

The purely mechanical nature of the differences between the streaming approach and a call/return approach strongly suggests that linguistic support could remove this inconvenience and make high-performance, flexible code the obvious and easy choice for developers.

## 9 Related Work

In the 1990s, the SFIO [14] library was introduced as a replacement for `stdio`. It introduced a number of advancements such as the ability to stream into a string buffer instead of a file descriptor, a feature later adopted by POSIX 2018 [5] in `stdio` using the `fmemopen()` call.

The standard libraries of Ruby [27], Python [22], Java [11], and C# all follow the problematic pattern we describe. Even Microsoft PowerShell [9], which allows piping objects between filters, uses a `toString()` method to return a string representation for basic textual output of objects. It also has a formatter pipeline, but does not use double dispatch.

Squeak Smalltalk's `Stream` class implements a `print:` message that's very similar to PWS's `writeObject:`, in that it double dispatches to the object using the `printOn:` message with the stream as argument [12]. Some, but not all of the implementors of `printOn:` then dispatch back to the stream using `print:`, for example for outputting elements of collections. However, there is no automatic generation of the stream writer message, no triple dispatch and no hierarchy of related streams connected via chaining.

C++'s `ostream` class also passes the stream to overloads of the << operator, making incremental output possible [15]. However, there is no triple dispatch and no chaining in order to support a hierarchy of related output streams, and there is no protection against cyclical object graphs.

The Common Lisp Input Manager (CLIM) [17] includes byte streams that can be connected directly to output windows without having to go through Unix pseudo terminals, similar to the special stream target discussed in Section 3.6. CLIM streams use CLOS [10] generic functions, so can make use of language-provided multiple dispatch facilities instead of having to use triple dispatch.

## 10 Summary and Outlook

This paper looked at the intersection of object-oriented and stream processing as embodied by Unix `stdio` from a software-architectural point of view. With that perspective, the standard method of adapting object-oriented languages to the pipes and filters was found to be lacking in several respects, particularly losing both the polymorphism of the object-oriented style and the streaming capabilities of the pipes and filters style, in a worst of both worlds situation.

The paper then introduces Polymorphic Write-Streams, which use dynamic language features to adapt between the two architectural styles while preserving the benefits of both: polymorphism and streaming performance.

This successful adaptation suggests that the same sort of combination of an architectural point of view with dynamic language capabilities could also yield a more object-oriented version of `stdin`, completing a standard object I/O library, *stdoio*.

## References

[1] 1994. *fwrite(3) BSD Library Functions Manual.*

[2] 2001. *PDF Reference: Adobe Portable Document Format Version 1.4 with Cdrom* (3rd ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[3] 2009. *printf(3) BSD Library Functions Manual.*

[4] 2012. NeXT Property Lists. https://en.wikipedia.org/wiki/Property_list

[5] 2013. Standard for Information Technology—Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7. *IEEE Std 1003.1, 2013 Edition (incorporates IEEE Std 1003.1-2008, and IEEE Std 1003.1-2008/Cor 1-2013)* (April 2013), 1–3906. https://doi.org/10.1109/IEEESTD.2013.6506091

[6] Apple Inc. 2009. Cocoa Reference. http://developer.apple.com/docs/Cocoa/

[7] Hans-J. Boehm, Russ Atkinson, and Michael Plass. 1995. Ropes: An Alternative to Strings. *Softw. Pract. Exper.* 25, 12 (Dec. 1995), 1315–1330. https://doi.org/10.1002/spe.4380251203

[8] S. R. Bourne. 1978. UNIX Time-Sharing System: The UNIX Shell. *Bell System Technical Journal* 57, 6 (1978), 1971–1990. https://doi.org/10.1002/j.1538-7305.1978.tb02139.x arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/j.1538-7305.1978.tb02139.x

[9] Hristo Deshev. 2008. *Pro Windows PowerShell (Pro).* APress.

[10] Richard P. Gabriel, Jon L. White, and Daniel G. Bobrow. 1991. CLOS: Integrating Object-oriented and Functional Programming. *Commun. ACM* 34, 9 (Sept. 1991), 29–38. https://doi.org/10.1145/114669.114671

[11] James Gosling, Bill Joy, and Guy L. Steele. 1996. *The Java Language Specification* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[12] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. *SIGPLAN Not.* 32, 10 (Oct. 1997), 318–326. https://doi.org/10.1145/263700.263754

[13] Daniel H. H. Ingalls. 1986. A Simple Technique for Handling Multiple Polymorphism. *SIGPLAN Not.* 21, 11 (June 1986), 347–349. https://doi.org/10.1145/960112.28732

[14] David Korn and Kiem phong Vo. 1991. SFIO: Safe/Fast String/File IO. In *In Proc. of the Summer '91 Usenix Conference.* 235–256.

[15] Angelika Langer and Klaus Kreft. 2008. *Standard C++ IOStreams and Locales: Advanced Programmer's Guide and Reference* (1st ed.). Addison-Wesley Professional.

[16] Doug McIlroy. 1964. Pipes Proposal. http://doc.cat-v.org/unix/pipes/

[17] Scott McKay. 1991. CLIM: The Common Lisp Interface Manager. *Commun. ACM* 34, 9 (Sept. 1991), 58–59. https://doi.org/10.1145/114669.114675

[18] Adobe Press. 1985. *PostScript Language Reference Manual* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

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

[20] Bjarne Stroustrup. 2000. *The C++ Programming Language* (3rd ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[21] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. 2009. *Software Architecture: Foundations, Theory, and Practice.* Wiley Publishing.

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

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

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

[25] Marcel Weiher. 2003. stsh. http://objective.st/Scripting

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

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