A Shell-like Model for General Purpose Programming

Jeanine Miller Adkisson
Tokyo Institute of Technology, Japan
jneen@jneen.net

Johannes Westlund
Tokyo Institute of Technology, Japan
KTH Royal Inst. of Technology, Sweden
jwestlun@kth.se

Hidehiko Masuhara
Tokyo Institute of Technology, Japan
masuhara@acm.org

Abstract

Shell scripting languages such as bash are designed to integrate with an OS, which mainly involves managing processes with implicit input and output streams. They also attempt to do this in a compact way that could be reasonably typed on a command-line interface.

However, existing shell languages are not sufficient to serve as general-purpose languages—values are not observable except in raw streams of bytes, and they lack modern language features such as lexical scope and higher-order functions.

By way of a new programming language, *Magritte*, we propose a general-purpose programming language with semantics similar to bash. In this paper, we discuss the early design of such a system, in which the primary unit of composition, like bash, is processes with input and output channels, which can be read from or written to at any time, and which can be chained together via a pipe operator. We also explore concurrency semantics for such a language.

CCS Concepts • **Software and its engineering → Scripting languages**; **Command and control languages**; *Concurrent programming languages*; *Data flow languages*.

ACM Reference Format:
Jeanine Miller Adkisson, Johannes Westlund, and Hidehiko Masuhara. 2019. A Shell-like Model for General Purpose Programming. In *Companion of the 3rd International Conference on Art, Science, and Engineering of Programming (Programming '19), April 1–4, 2019, Genova, Italy.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3328433.3328444

1 Motivation

The UNIX shell programming model has played an important role in integrating applications and operating systems by composing programs—spawning independent programs in parallel to communicate over operating-system pipes. Beyond the most simple tasks, however, bash and similar tools break down, due to various language deficiencies.

In this paper, we use the term *pipe-based language* to mean a programming language intended to be used on a command line, with the ability to spawn many processes which communicate through synchronous channels or pipes in an ad-hoc manner—and in which this facility is the primary method of composing different functions or units.

Our overarching goal is to create a language and a programming system that retains the pipe-based programming and interaction model of bash, but allows for large programs in a way that existing shell languages do not.

Outline

In this work, we present a new pipe-based language *Magritte*. Section 2 describes design requirements for a language in this space. Section 3 and 4 describe Magritte's design and implementation. Section 5 describes our future goals with this project, and in Section 6 we review various related work and alternative approaches.

2 Design Considerations

In this section we discuss several feature requirements and design considerations for a pipe-based language to be viable for large programs.

2.1 Programming With Values

2.1.1 Value Pipes

It is desirable in a general purpose language to be able to create and use complex data structures, and to use them freely throughout the language. In particular, if we are to allow for pipe-based composition to be the core composition method for large programs, we must allow rich values to be passed through pipes.

Several projects[1, 4], including bash itself, have attempted to add arrays and associative arrays as standard objects, but not as first class values: they do not allow these data structures to be passed into or returned from functions, or importantly, passed through pipes.

Furthermore, serialization through pipes is generally not possible except in a single-threaded case, since complex data can be corrupted through interleaving.

For example, consider the fairly common architecture of producer/consumer: A *producer* process produces values that are processed in parallel by multiple *consumer* processes, and the values are collected in a single output. This might be expressed in bash as:

```
# consume the stream, labeling every line
label() { while read x; do echo "$1$x"; done ;}

# process the stream with two threads
split() { label a & label b & ;}

# produce and process the numbers 1-100,
# limiting output to the first 3 lines
seq 100 | split | head -3
```

With this code, a user might expect the output to be three lines, each consisting of a letter a or b, and a number, for example:

*This work was presented at the 122nd IPSJ SIGPRO workshop on January 17, 2019. The presentation was not refereed, and the same manuscript was distributed only to the participants to the workshop.

Programming '19, April 1–4, 2019, Genova, Italy

ACM ISBN 978-1-4503-6257-3/19/04…$15.00
https://doi.org/10.1145/3328433.3328444

```
a1
a2
a3
```

```
b1
a3
b2
```

```
a2
b1
a3
```

Unfortunately, when we run this process, the outputs from the `label` function become interleaved[1], resulting in outputs such as:

```
a2
b1
b
```

```
b12
a
a4
```

```
a
b12
a34
```

This behavior is in accordance with the Linux User's Manual [7]:

> The communication channel provided by a pipe is a *byte stream*: there is no concept of message boundaries.

In order to provide a channel implementation that is usable for large programs, we must ensure that it is *consistent*—that is, that every read corresponds to exactly one write. Value pipes accomplish this by making a value the smallest atomic unit of communication.

2.1.2 Capture and Substitution

For processes that output values, we need to support a mechanism to capture those values for use in variables, data structures, and function arguments, similar to backticks or `$(...)` in bash. This enables processes to be used like functions, which return data to their caller.

But since in most shells, rich values are not writable to output streams, the output cannot effectively be used as a return path for values. Es Shell[4] accounts for this by introducing a *return value* which is separate from stream output and can be accessed via special-purpose call syntax.

If we allow values to be written to output streams, however, we can directly capture values written to the output.

2.1.3 Modern Language Features

Users will expect a modern programming language to have:

- *Lambda functions with closure*. This requires the introduction of lexically scoped variables.
- *Dynamic variables*. Most shell languages already include these, as OS Environment variables are dynamic by nature.
- *Product structures with support for open recursion*. a prototype-based object system is sufficient for this.
- *Sum structures*. In an untyped language, a method for pattern matching over nestable heterogenous lists is sufficient.

2.2 Automatic Process Cleanup

2.2.1 Interruption

In shell programming, we tend to compose infinitely running processes together as pipeline elements. Thus we require a well-defined semantics of process *interruption*[2]: automatic clean-up of processes that will no longer be used. Consider the following Magritte code:

```
read-lines tmp/large-file                          1
  | take 10                                        2
  | each (?line => do-expensive-work $line)        3
```

In this example, three processes are spawned concurrently: (1) a process with an open file that writes one line at a time to its output, (2) a process that reads 10 times from its input, writes each entry to the output, and then exits, and (3) a process that reads every input and calls a function to perform an expensive task. A user's intent when typing such code may be to read 10 lines from a file and synchronously perform an action on each line.

A user will also expect that, after the first 10 lines are processed and the `take` function returns, the file will be closed, and all three processes exit. This expectation is despite the fact that process (1) is specified to read the entire file, and process (3) is an infinite loop.

In a naive implementation using synchronous channels, process (1) will never be able to write more than 10 lines, and will remain blocked on its output with the file open forever. Similarly, the call to the `each` function will never be notified that its input has finished, and will block forever on its input stream.

2.2.2 Compensation

In interacting with an operating system, it is necessary to mange side effects, and gracefully recover or restore state in the case of an interruption. Such an error-handling system would also be a way for user code to directly observe interruption.

Without compensation of errors, the only way of observing forever-blocked processes would be to inspect the process table, or to observe the memory footprint of the program. Thus, we can define the primary task of the interruption system as *running a process's compensation actions at the appropriate time*.

2.2.3 Lazy Interruption vs. Eager Interruption

In a system such as this, there is a language design choice that must be made—whether interruption is *eager*, in which processes are interrupted immediately upon closing a channel, or *lazy*, in which processes are only interrupted upon interaction with a closed channel. Both approaches have advantages and drawbacks. In making this choice, we desire that the behavior of interruption be *predictable*—however, there are two competing viewpoints for predictability.

Consider the following example, with the assumption that `do-other-operation` does not write to the standard output:

```
( put 1 2 3; do-other-operation ) | take 3
```

The left-hand side of the pipe will write three times, and then continue to do other processing in-thread that does not write to the standard output. The final `take 3` operation will return after 3 inputs are read. It is important to decide, then, whether the process on the left should be interrupted in the middle of `do-other-operation` (*eager interruption*), or whether it should be left alone until it attempts to write a value (*lazy interruption*).

From the perspective of someone spawning a process, eager interruption can seem more predictable, as they can guarantee a point at which the process has stopped doing work.

[1] We have observed some behavior in the output that cannot be explained simply by interleaving, suggesting there may be some other race conditions in play.

[2] We use the term *interruption* in the generic sense—to mean the halting of normal flow in a process, due to some external event. It is unrelated to hardware or OS signalling.

However, from the perspective of a function author, lazy interruption is more predictable, because the author can identify precisely which points in the code have the potential to be interrupted—those points which run a `put` or a `get`. Contrast this with eager-interruption semantics, where any point in the code may be interrupted, introducing the need for users to either mark critical sections and be very careful with implementing stateful algorithms.

On the other hand, lazy interruption has the disadvantage that a producing process must do enough work to produce one more value than will be consumed. If each value is relatively cheap to produce, this is not a problem, but in the case that the values are expensive to produce, this would result in a large amount of unnecessary work.

2.2.4 Closing of Channels

Given that multiple processes may be reading from and writing to a channel, it is often the case that a communicating process will end when there are other processes still communicating over the channel. It would not be appropriate in this case to interrupt other processes attached to the channel, as they are still able to communicate.

We define our guiding principle for the appropriate time to interrupt a process as: *A process is interrupted exactly when it can no longer be woken up*. When a process is blocked on a synchronous channel, it will be woken up as soon as another process communicates on the other end. Therefore the appropriate time for it to be interrupted is when it is blocked on a channel that will never receive any more operations.

This can be difficult to detect when a reference to a channel can be passed anywhere in the program as a standard value. Luckily, the arrangement of pipes and channels are usually specified at process spawn time. We can therefore relax our constraint to guarantee that channels will be closed at appropriate times in *common architectures*, and that they never close if there are active readers or writers.

2.2.5 Reopening Channels

Some systems, like UNIX named pipes, allow a channel to be reused by new processes after it has been closed[7]. This is, however, not a desirable feature, as it can lead to some unexpected races between a channel closing and a new process spawning. Consider the example:

```
c = (make-channel) # create a new channel
& count-forever > $c # write infinitely
& take 10 < $c # read 10 elements and exit
put 10 > $c # write once from a new process
```

This example represents an unavoidable race with first-class channels: between `take 10` closing the channel and `put 10` opening the channel for writing. If we allow channel reopening, we will either block forever, or insert the number 10 into the stream, depending on which happens first. However if we do not allow channel reopening, we can say that, if a process initiates a read or write on a closed channel, it is immediately interrupted. In this case, both sides of the former race have the same termination behavior—the process is closed when `take 10` returns.

2.2.6 Masking Interruptions

We must include a facility to control the extend of interruptions. This is because an interruption semantics can make it difficult to perform final calculations after a channel is closed.

Consider the following example, which sums all numbers from the standard input:

```
(sum) = (
  total = 0
  each (?x => %total = (add %total %x))
  put %total
)
```

In this example, the function `each` will consume the input stream and mutate a lexical variable (marked with `%`). After the entire input stream is consumed, we wish to output the resulting `%total` value.

However, with the semantics described above, the `each` function will loop until it receives an interrupt signal from the input channel closing, which will interrupt the entire process and not continue to the following line.

3 Description of Magritte

We propose a language that meets the above requirements.

3.1 Values and Variables

3.1.1 Lexical vs. Dynamic Variables

In order to support both lexical and dynamic variables, we introduce a separate syntax for lexically scoped variables at variable reference and mutation points, using a `%` instead of `$` (for example, `%x`). Let-binding is left unadorned (`x = 1`), and will bind a variable both lexically and dynamically.

Let-binding uses this plain syntax to maintain compatibility with standard environment files. Variable binders in lambda arguments use `?` (e.g. `?x`) to allow for unambiguous pattern matching, and are also bound both lexically and dynamically.

For mutation, we use assignment syntax, but with the location (dynamic or lexical) specified on the left-hand side, e.g. `%x = 1` or `$x = 1`. In this way we avoid the need to declare variables, and allow ourselves to throw an exception when trying to mutate a non-existent variable, rather than silently creating a new local binding.

3.1.2 Lambda Functions

Lambda functions are specified with parenthesized expressions containing the arrow symbol `=>`, which separates the bindings from the body. Multiple matching clauses are possible, and can match simple patterns. Clauses are separated at the beginning of lines that contain `=>` (where "lines" are also terminated by `;`, and do not consider nested newlines), which avoids the need for any indentation-based parsing. For example:

```
my-function = (
  ?x =>
    put one-argument %x
  ?y ?z =>
    put two-arguments
    put %y %z
  ... => ...
)
```

Named functions can also be defined using a parenthesized expression as the left hand side of an assignment:

```
(my-function ?x ?y) = ...
```

```
# equivalent to
my-function = (?x ?y => ...)
```

Functions maintain the scope of any free lexical variables in the body, including for mutation.

3.1.3 Nestable Vectors

We extend the concept of an `argv` vector such that it is nestable, using the compact syntax `[ ]`. Combined with bareword strings (string literals without quotation), we can represent trees in a straightforward manner:

```
a-tree = [node [node [leaf 1] [leaf 2]] [leaf 3]]
```

These can be matched in lambda arguments by patterns such as `[node ?x]`. A typical strategy for traversing this kind of structure might be a function that puts all leaf node values to its output in a predefined order, to be consumed by another process.

The builtin function `for` takes a vector as an argument and outputs each element in order, so that a vector can be traversed with:

```
for [1 2 3 4 5] | each (?el => ...)
```

3.1.4 Environments as Objects

Variable environments can be captured directly as key-value maps with a parent pointer. Environment capture uses `{ }` syntax to run a block of code, then remove the running environment from its parent list. The resulting environment value is substituted in place of the `{ }` expression. Therefore all assignment syntaxes are available, including function definition. For example:

```
(make-account ?balance) = {
 balance = $balance
 (deposit ?amt) = (%balance = (add %amt %balance))
 (withdraw ?amt) = (%balance = (sub %amt %balance))
}
```

A planned extension would allow using special syntax to register additional parents, allowing users to inherit by direct delegation.

Environment lookup uses the `!` symbol, as in `$env!key`. We use this symbol because it is already reserved by bash, unlike both `/` or `.` which need to be available in bareword syntax to indicate file paths for external programs. Environments can also be mutated using the same access syntax. For example:

```
account = (make-account 10)
$account!withdraw 4
put $account!balance # => 6
```

3.1.5 Collection and Substitution

A priori, a capture mechanism such as described in Section 2.1.2 would have to return a list, as any process may output zero or more values. However, as a function calling convention, that would require callers to manually unwrap lists on every function call.

In order to simplify substitution, Magritte uses normal parentheses to collect and *expand* the resulting values into the current command vector—increasing the argument number by the number of values output from the function. For example:

```
# A function defintion: output three values
(count-three) = (put 1; put 2; put 3)

# Collect three writes and expand 1 2 3 in-place
other-fn 0 (count-three) 4

# equivalent to
other-fn 0 1 2 3 4
```

These semantics are similar to the `$(...)` syntax in bash, with the exception that we have the ability to properly separate values without relying on whitespace.

This mechanism is also available in vector literals, allowing us to collect outputs as a vector:

```
outputs = [(some-command)]
```

The `list` built-in function, which simply returns its argument vector, is also available for this purpose. The `for` function mentioned above can be used to splat vector arguments into function calls:

```
some-fn $arg1 $arg2 (for [$arg3 $arg4])
```

3.1.6 Blocks

A parenthesized grouping that is *not* in argument position is a *block*. Blocks do not collect or modify the environment's channels in any way, but instead simply run the code contained within them, and output values normally. These are mostly used to group commands within a pipeline, and in the body of function definitions:

```
generate-values | (process; process; process)
```

In the case that a user might want to use a substitution at the root level—i.e. to generate a function and immediately call it, we provide the `exec` builtin which executes its arguments as a command.

3.2 Channels and Processes

3.2.1 Synchronous Channels

Channels in Magritte are *synchronous*—readers and writers cannot continue until a communication is completed successfully. Given the decision to allow rich values to be passed through channels, we have decided that a buffer is not as necessary for performance purposes, since a single write may contain an arbitrarily large amount of data—or for that matter a process handle or object reference. Additionally, synchronous channels have simpler semantics both for implementation and for users, and we leave open the possibility of user-implemented queues, such as:

```
producer | buffer 10 | consumer
```

3.2.2 Spawning and Redirecting

Spawning uses an ampersand (&), similar to bash, at the *beginning* of a command indicating that it should be spawned in a new thread.

Standard input and output may be redirected into and from channels using the > and < symbols, or chained together with the pipe (|) symbol, much like `bash`. Commands in pipelines will all be spawned in their own threads, with the exception of the final command, which will be run in the current process.

3.3 Interruption and Compensation

3.3.1 Lazy Interruption

We have decided that the predictability of lazy interruption is worth the tradeoff for the extra-values problem discussed in Section 2.2.3, and we discuss some ways to mitigate this problem in Section 5.

3.3.2 Process Registration

In order to satisfy the constraints of Section 2.2.4, we maintain a process register inside of each channel, so that we can decide when all readers or all writers have returned or been interrupted, at which time we can guarantee that processes on the other side of the channel cannot be woken up *without spawning new processes*. Therefore this covers the architecture of pipelines, in which many processes are spawned together in fixed configurations. Other architectures will have to manually manage process shutdown in some cases.

3.3.3 Compensation and Unconditional Compensation

We employ a variant of the compensation mechanism introduced by Inoue et al.[6] using the `%%` operator to indicate a compensation action, which is run in case of an interruption in the left hand side or subsequent lines. Compensations are cleared at the end of the current function body:

```
(my-function) = (
  action %% cleanup-action
  # in effect until the end of the function
)
```

Additionally, we define *unconditional compensations* using the `%%!` operator, which run both in the case of an interruption and in the case of a normal return. In this way, they are analogous to `finally` or `ensure` sections of standard exception handling. For example, the function `read-lines` above could be implemented as:

```
(read-lines ?fname) =
 (f = (open-file $fname) %%! close-file $fname
  until (=> eof? $f) (=> read-until "\n" $f))
```

In this way, we can ensure that the file is closed when the function exits, whether by a normal return or by interruption.

3.3.4 Interrupt Handling

While we plan to explore more general mechanisms for exception handling, we find that it suffices for most applications to provide two builtin functions, `produce` and `consume`, to indicate the intent to fill or consume the entirety of the output or input streams, respectively. Each of these functions takes a single zero-argument function which will loop forever until the standard output or standard input respectively is closed, whereby control flow continues after the invocation. Using these functions, we might define `each` as:

```
(each ?fn) = (consume (=> %fn (get)))
```

With this definition, the call to `consume` will mask the interruption from the standard input closing, and control flow after any `each` invocation will continue as normal. This is enough to resolve the issue discussed in Section 2.2.6.

4 Implementation of Magritte

The current version of Magritte is implemented as a straightforward interpreter written in Ruby, using Ruby's builtin threads, exceptions, and mutexes to implement processes and channels.

4.1 Channel Registry

Our channel implementation is a standard implementation of synchronous channels, with the addition of four intrinsic methods, used only internally by the interpreter: `add_reader`, `remove_reader`, `add_writer`, and `remove_writer`, which register and deregister processes as described in Section 3.3.2. When a `remove_*` method results in an empty set, it will additionally close the channel and raise an internal exception in every blocked thread.

Once the channel is closed, every call to `read` and `write` will interrupt the calling process as described in Section 2.2.5.

In order to reduce unnecessary use of Ruby threads, we also find it is simpler, instead of registering *processes* to the channels, to register *stack frames*. In this way, we can register all inputs and outputs on frame entry, and use standard Ruby exception handling to ensure we properly run compensations and deregister inputs and outputs on frame exit. This means that different frames in the same process can be connected to different channels, which makes the > and < redirection syntax straightforward to implement—we simply push a new frame with different channels attached.

Interruptions then cascade naturally—when a channel closes, a process is interrupted, causing it to unwind its stack and deregister channels, thereby potentially causing other channels to close.

4.2 Spawning Order Dependency

It is necessary to take some care with the implementation of the spawning primitive (&), that we wait until the spawned process has finished registering its channels before the spawning process continues. Consider the following example, which outputs 10 numbers, possibly out of order:

```
(drain) = (each (?x => put %x))
count-forever | (& drain; & drain) | take 10
```

The middle process is responsible for spawning two processes that funnel data from their input to their output. However, since they are both spawned in the background, the spawning process will immediately return. In general, the `drain` processes should be keeping the two pipes open. However, if we do not take care to wait until they have finished registering their channels, there is a risk that the spawning process will return first and close the two pipes.

4.3 Collectors

In order to implement the return semantics described in Section 2.1.2, we also implement a write-only channel called a *collector*, and an intrinsic that waits for channel closing. Collectors cannot be created directly by users, but only appear in the interpreter when we evaluate parentheses in argument position.

Naively, collectors would ignore registration commands and simply append written elements to an array. However, it is still necessary to track registered writers. Consider the example:

```
(range ?n) = (count-forever | take %n)
my-list = [(range 10 | (& drain; & drain))]
```

In this example, there is an open question of how long we should wait until reading the collection and continuing. If we naively wait until the base command is finished, we will continue early and miss values that may be written later. Thus substitution waits until all writers to the collector have deregistered.

To implement this, we add a `wait_for_close` intrinsic to collectors, which returns immediately if closed, and otherwise adds the current thread to a waiting set and sleeps. Upon closing the channel, all elements of this waiting set are awoken, and flow continues. Thus the algorithm for substituting a block is:

```
* Create a new collector $c
* Run the parenthesized expressions with ↩
    standard output set to the collector
* Run $c.wait_for_close
```

In the most common case, there will only be a single thread writing to the collector, so that the channel will be closed by the time the evaluation is finished, making `wait_for_close` a null operation.

5 Discussion

5.1 Open Checking

Because our system implements *lazy interruption*, we must mitigate the problem of producers creating one extra element than is needed, in the case that elements are expensive to produce. One possibility is to introduce the concept of *open checking*, so that producer processes can periodically run a builtin command that will interrupt if the output is closed, without actually writing any data to the channel.

5.2 OS Integration

In order to be viable as a shell, we must integrate seamlessly with a POSIX-like environment. This involves marshalling values to strings and byte streams in order to communicate with external processes, and allowing external processes to interact with Magritte values. We plan to use a suite of parsing and unparsing functions for linewise and table data, along with a custom JSON streaming format to allow integration with most languages. We also plan to use a socket and a static executable to allow external programs to call back into Magritte lambda functions.

5.3 Performance

The current Ruby interpreter is slow. We plan to bootstrap using a JIT-compiled virtual machine implemented in RPython.

5.4 Desktop Scripting

We believe Magritte offers a promising solution to the *desktop scripting problem*: an interface for end users to integrate and automate independent programs, a problem attempted by projects such as Guile Scheme[12] and TCL[9]. What differentiates our approach is that our platform should be able to integrate with programs as they are currently written, without the need for further integration on the part of application developers.

6 Related Work

6.1 General-Purpose Languages

Many general purpose languages, including Go[3] and Clojure[5], contain robust channel implementations, which can be used to implement a variety of concurrent algorithms. In such a system we could implement something similar to our system by adding input and output channels to every function and heavily currying functions.

However, such a system would still require manual use of input and output channels when a simple pipeline would suffice—and the user must still translate between returning vs. outputting functions and expressions. Additionally, these systems do not contain any kind of channel closing semantics or automatic process cleanup.

6.2 Shell Extensions

Many systems, such as `xonsh`[11] and Es Shell[4], attempt to extend traditional shells with object or functional semantics, but do not sufficiently integrate their semantics with pipes, leaving pipes to remain bytes-only.

Systems that extend existing languages with shell semantics, such as `scheme-shell`, also do not integrate fully with pipes.

6.3 Object Shells

Projects such as `powershell`[8] and `mash`[10] allow objects (.NET objects and Scala objects, respectively) to be passed through pipelines. However, this approach still ties the language to one specific platform or object system and makes it difficult to integrate with programs written for other platforms. By *only* relying on the most universal OS concepts of standard input/output, argument vectors, and interrupt signals, we should be able to integrate programs across a wide variety of languages and platforms.

6.4 PUSH Shell

The PUSH shell[2] extends a traditional shell with *fan-out* and *fan-in* operators which separate data into discrete records. This enables it to be used safely to orchestrate distributed computing tasks. However, the underlying pipe implementation remains byte-based, so that features like Magritte's collection and expansion remain impossible.

7 Conclusion

We proposed Magritte, which is both capable of integrating closely with operating systems, and serving as a general-purpose language for larger applications. Our channel implementation allows for robust composition of concurrent algorithms, and can also serve as the fundamental unit of composition for our language, which integrates with modern general-purpose language features like rich data structures and lambda functionsn with lexical scoping.

At the same time, the language stays close enough to the common OS spawning model that we believe it should be straightforward to integrate with an operating system. Because of this, we believe Magritte is capable of filling a currently unserved niche as a desktop integration language, and also provides an interesting platform for experimenting and building concurrent algorithms.

Acknowledgments

This work was supported by JSPS KAKENHI Grant Number 18H03219. J. Westlund's contribution to this research was made possible by The Sweden-Japan Foundation and Stockholms Grosshandelssocietet.

References

[1] Tom Duff. 1990. RC—a Shell for Plan 9 and UNIX. In *UNIX Vol. II*, A. G. Hume and M. D. McIlroy (Eds.). W. B. Saunders Company, Philadelphia, PA, USA, 283–296.

[2] Noah Paul Evans and Eric Van Hensbergen. 2009. Brief Announcement: PUSH, a DISC Shell. In *Proceedings of the 28th ACM Symposium on Principles of*

Distributed Computing (PODC '09). ACM, New York, NY, USA, 306–307. https://doi.org/10.1145/1582716.1582780

[3] Robert Griesemer, Rob Pike, and Ken Thompson. 2018. The Go Programming Language. Retrieved Dec 2018 from *http://golang.org/*.

[4] Paul Haahr and Byron Rakitzis. 1993. Es: A shell with higher-order functions. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings (USENIX'93)*. USENIX Association, Berkeley, CA, USA, 53–62.

[5] Richard Hickey. 2008. The Clojure Programming Language. In *Proceedings of the 2008 Symposium on Dynamic Languages*. ACM, New York, NY, USA.

[6] Hiroaki Inoue, Tomoyuki Aotani, and Atsushi Igarashi. 2018. ContextWorkflow: A Monadic DSL for Compensable and Interruptible Executions. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018) (Leibniz International Proceedings in Informatics (LIPIcs))*, Todd Millstein (Ed.), Vol. 109. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2:1–2:33. https://doi.org/10.4230/LIPIcs.ECOOP.2018.2

[7] Linux man-pages Project. 2018. *pipe(7) Linux User's Manual* (4.16 ed.). The Linux Foundation, https://www.kernel.org/doc/man-pages/.

[8] Microsoft. 2018. PowerShell Documentation. Retrieved Dec 2018 from *https://docs.microsoft.com/en-us/powershell/*.

[9] John K. Ousterhout. 1989. *Tcl: An Embeddable Command Language*. Technical Report. Berkeley, CA, USA.

[10] Matt Russell. 2018. Mash: An object shell for UNIX. Retrieved Dec 2018 from *http://mash-shell.org/*.

[11] Anthony Scopatz. 2018. The Xonsh Shell. Retrieved Dec 2018 from *https://xon.sh/*.

[12] The GNU Project. 2018. The Guile Programming Language. Retreived Dec 2018 from *http://gnu.org/s/guile*.