The Inferno Shell

Roger Peppé
rog@vitanuova.com

ABSTRACT

The Inferno shell *sh* is a reasonably small shell that brings together aspects of several other shells along with Inferno's dynamically loaded modules, which it uses for much of the functionality traditionally built into the shell. This paper focuses principally on the features that make it unusual, and presents an example network chat application written entirely in *sh* script.

Introduction

Shells come in many shapes and sizes. The Inferno shell *sh* (actually one of three shells supplied with Inferno) is an attempt to combine the strengths of a Unix-like shell, notably Tom Duff's *rc*, with some of the features peculiar to Inferno. It owes its largest debt to *rc*, which provides almost all of the syntax and most of the semantics too; when in doubt, I copied *rc*'s behaviour. In fact, I borrowed as many good ideas as I could from elsewhere, inventing new concepts and syntax only when unbearably tempted. See Credits for a list of those I could remember.

This paper does not attempt to give more than a brief overview of the aspects of *sh* which it holds in common with Plan 9's *rc*. The reader is referred to *sh*(1) (the definitive reference) and Tom Duff's paper "Rc – The Plan 9 Shell". I have occasionally pinched examples from the latter, so the differences are easily contrasted.

Overview

Sh is, at its simplest level, a command interpreter that will be familiar to all those who have used the Bourne-shell, C shell, or any of the numerous variants thereof (e.g. *bash*, *ksh*, *tcsh*). All of the following commands behave as expected:

```
date
cat /lib/keyboard
ls -l > file.names
ls -l /dis >> file.names
wc <file
echo [a-f]*.b
ls | wc
ls; date
limbo *.b &
```

An *rc* concept that will be less familiar to users of more conventional shells is the rôle of *lists* in the shell. Each simple *sh* command, and the value of any *sh* environment variable, consists of a list of words. *Sh* lists are flat, a simple ordered list of words, where a word is a sequence of characters that may include white-space or characters special to the shell. The Bourne-shell and its kin have no such concept, which means that every time the value of any environment variable is used, it is split into blank separated words. For instance, the command:

```
x='-l /lib/keyboard'
ls $x
```

would in many shells pass the two arguments `-l` and `/lib/keyboard` to the `ls` command. In *sh*, it will pass the single argument `-l /lib/keyboard`.

The following aspects of *sh*'s syntax will be familiar to users of *rc*.

File descriptor manipulation:

```
echo hello, world > /dev/null >[1=2]
```

Environment variable values:

```
echo $var
```

Count number of elements in a variable:

```
echo $#var
```

Run a command and substitute its output:

```
rm `{grep -li microsoft *}
```

Lists:

```
echo (((a b) c) d)
```

List concatenation:

```
cat /appl/cmd/sh/^(std regex expr)^.b
```

To the above, *sh* adds a variant of the `` `{} `` operator: `"{}`, which is except that it does not split the input into tokens, for example:

```
for i in "{echo one two three} {
    echo loop
}
```

will only print `loop` once.

Sh also adds a new redirection operator `<>`, which opens the standard input (by default) for reading *and* writing.

Command blocks

Possibly *sh*'s most significant departure from the norm is its use of command blocks as values. In a conventional shell, a command block groups commands together into a single syntactic unit that can then be used wherever a simple command might appear. For example:

```
{
    echo hello
    echo goodbye
} > /dev/null
```

Sh allows this, but it also allows a command block to appear wherever a normal word would appear. In this case, the command block is not executed immediately, but is bundled up as if it was a single quoted word. For example:

```
cmd = {
    echo hello
    echo goodbye
}
```

will store the contents of the braced block inside the environment variable `$cmd`. Printing the value of `$cmd` gets the block back again, for example:

```
echo $cmd
```

gives

```
{echo hello;echo goodbye}
```

Note that when the shell parsed the block, it ignored everything that was not syntactically relevant to the execution of the block; for instance, the white space has been reduced to the minimum necessary, and the newline has been changed to the functionally identical semi-colon.

It is also worth pointing out that echo is an external module, implementing only the standard *Command* (2) interface; it has no knowledge of shell command blocks. When the shell invokes an external command, and one of the arguments is a command block, it simply passes the equivalent string. Internally, built in commands are slightly different for efficiency's sake, as we will see, but for almost all purposes you can treat command blocks as if they were strings holding functionally equivalent shell commands.

This equivalence also applies to the execution of commands. When the shell comes to execute a simple command (a sequence of words), it examines the first word to decide what to execute. In most shells, this word can be either the file name of an external command, or the name of a command built into the shell (e.g. exit).

Sh follows these conventional rules, but first, it examines the first character of the first word, and if it is an open brace ({) character, it treats it as a command block, parses it, and executes it according to the normal syntax rules of the shell. For the duration of this execution, it sets the environment variable $* to the list of arguments passed to the block. For example:

```
{echo $*} hello world
```

is exactly the same as

```
echo hello world
```

Execution of command blocks is the same whether the command block is just a string or has already been parsed by the shell. For example:

```
{echo hello}
```

is exactly the same as

```
'{echo hello}'
```

The only difference is that the former case has its syntax checked for correctness as soon as the shell sees the script, whereas if the latter contained a malformed command block, a syntax error will be raised only when it comes to actually execute the command.

The shell's treatment of braces can be used to provide functionality similar to the eval command that is built into most other shells.

```
cmd = 'echo hello; echo goodbye'
'{'^$cmd^'}'
```

In other words, simply by surrounding a string by braces and executing it, the string will be executed as if it had been typed to the shell. Note the use of the caret (^) string concatenatation operator. *Sh* does provide free carets in the same way as *rc*, so in the previous example

```
'{'$cmd'}'
```

would work exactly the same, but generally, and in particular when writing scripts, it is good style to make the carets explicit.

Assignment and scope

The assignment operator in *sh*, in common with most other shells, is =.

```
x=a b c d
```

assigns the four element list (a b c d) to the environment variable named x. The value can later be extracted with the $ operator, for example:

```
echo $x
```

will print

```
a b c d
```

Sh also implements a form of local variable. An execution of a braced block command creates a new scope for the duration of that block; the value of a variable assigned with := in that block will be lost when the block exits. For example:

```
x = hello
{x := goodbye }
echo $x
```

will print hello. Note that the scoping rules are *dynamic* – variable references are interpreted relative to their containing scope at execution time. For example:

```
x := hello
cmd := {echo $x}
{
     x := goodbye
     $cmd
}
```

will print goodbye, not hello. For one way of avoiding this problem, see Lexical binding below.

One late, but useful, addition to the shell's assignment syntax is tuple assignment. This partially makes up for the lack of list indexing primitives in the shell. If the left hand side of the assignment operator is a list of variable names, each element of the list on the right hand ride is assigned in turn to its respective variable. The last variable mentioned gets assigned all the remaining elements. For example, after:

```
(a b c) := (one two three four five)
```

a is one, b is two, and c contains the three element list (three four five). For example:

```
(first var) = $var
```

knocks the first element off $var and puts it in $first.

One important difference between *sh*'s variables and variables in shells under Unix-like operating systems derives from the fact that Inferno's underlying process creation primitive is *spawn*, not *fork*. This means that, even though the shell might create a new process to accomplish an I/O redirection, variables changed by the sub-process are still visible in the parent process. This applies anywhere a new process is created that runs synchronously with respect to the rest of the shell script – i.e. there is no chance of parallel access to the environment. For example, it is possible to get access to the status value of a command executed by the `{} operator:

```
files=`{du -a; dustatus = $status}
if {! ~ $dustatus ''} {
     echo du failed
}
```

When the shell does spawn an asynchronous process (background processes and pipelines are the two occasions that it does so), the environment is copied so changes in one process do not affect another.

Loadable modules

The ability to pass command blocks as values is all very well, but does not in itself provide the programmability that is central to the power of shell scripts and is built in to most shells, the conditional execution of commands, for instance. The Inferno shell is different; it provides no programmability within the shell itself, but instead relies on external modules to provide this. It has a built in command `load` that loads a new module into the shell. The module that supports standard control flow functionality and a number of other useful tidbits is called `std`.

```
load std
```

loads this module into the shell. `Std` is a Dis module that implements the `Shellbuiltin` interface; the shell looks in the directory `/dis/sh` for the module file, in this case `/dis/sh/std.dis`.

When a module is loaded, it is given the opportunity to define as many new commands as it wants. Perhaps slightly confusingly, these are known as built-in commands (or just *builtins*), to distinguish them from commands executed in a separate process with no access to shell internals. Built-in commands run in the same process as the shell, and have direct access to all its internal state (environment variables, command line options, and state stored within the implementing module itself). It is possible to find out what built-in commands are currently defined with the command `loaded`. Before any modules have been loaded, typing

```
loaded
```

produces:

```
builtin   builtin
exit      builtin
load      builtin
loaded    builtin
run       builtin
unload    builtin
whatis    builtin
${builtin}          builtin
${loaded}           builtin
${quote}            builtin
${unquote}          builtin
```

These are all the commands that are built into the shell proper; I'll explain the `${}` commands later. After loading `std`, executing `loaded` produces:

```
!         std
and       std
apply     std
builtin   builtin
exit      builtin
flag      std
fn        std
for       std
getlines            std
if        std
load      builtin
loaded    builtin
```

```
or       std
pctl     std
raise    std
rescue   std
run      builtin
status   std
subfn    std
unload   builtin
whatis   builtin
while    std
~        std
${builtin}        builtin
${env}   std
${hd}    std
${index}          std
${join}  std
${loaded}         builtin
${parse}          std
${pid}   std
${pipe}  std
${quote}          builtin
${split}          std
${tl}    std
${unquote}        builtin
```

The name of each command defined by a loaded module is followed by the name of the module, so you can see that in this case std has defined commands such as if and while . These commands are reminiscent of the commands built in to the syntax of other shells, but have no special syntax associated with them: they obey the normal argument gathering and execution semantics.

As an example, consider the for command.

```
for i in a b c d {
    echo $i
}
```

This command traverses the list (a b c d) executing {echo $i} with $i set to each element in turn. In *rc*, this might be written

```
for (i in a b c d) {
    echo $i
}
```

and in fact, in *sh*, this is exactly equivalent. The round brackets denote a list and, like *rc*, all lists are flattened before passing to an executed command. Unlike the for command in *rc*, the braces around the command are not optional; as with the arguments to a normal command, gathering of arguments stops at a newline. The exception to this rule is that newlines within brackets are treated as white space. This last rule also applies to round brackets, for example:

```
(for i in
    a
    b
    c
    d
    {echo $i}
)
```

does the same thing. This is very useful for commands that take multiple command block arguments, and is actually the only line continuation mechanism that *sh* provides (the usual backslash (\) character is not in any way special to *sh*).

Control structures

Inferno commands, like shell commands in Unix or Plan 9, return a status when they finish. A command s status in Inferno is a short string describing any error that has occurred; it can be found in the environment variable $status. This is the value that commands defined by std use to determine conditional execution – if it is empty, it is true, otherwise false. Std defines, for instance, a command ~ that provides a simple pattern matching capability. Its first argument is the string to test the patterns against, and subsequent arguments give the patterns, in normal shell wildcard syntax; its status is true if there is a match.

```
~ sh.y '*.y'
~ std.b '*.y'
```

give true and false statuses respectively. A couple of pitfalls lurk here for the unwary: unlike its *rc* namesake, the patterns *are* expanded by the shell if left unquoted, so one has to be careful to quote wildcard characters, or escape them with a backslash if they are to be used literally. Like any other command, ~ receives a simple list of arguments, so it has to assume that the string tested has exactly one element; if you provide a null variable, or one with more than one element, then you will get unexpected results. If in doubt, use the $" operator to make sure of that.

Used in conjunction with the $# operator, ~ provides a way to check the number of elements in a list:

```
~ $#var 0
```

will be true if $var is empty.

This can be tested by the if command, which accepts command blocks for its arguments, executing its second argument if the status of the first is empty (true). For example:

```
if {~ $#var 0} {
     echo '$var has no elements'
}
```

Note that the start of one argument must come on the same line as the end of of the previous, otherwise it will be treated as a new command, and always executed. For example:

```
if {~ $#var 0}
     {echo '$var has no elements'}      # this will always be executed
```

The way to get around this is to use list bracketing, for example:

```
(if {~ $#var 0}
     {echo '$var has no elements'}
)
```

will have the desired effect. The if command is more general than *rc* s if, in that it accepts an arbitrary number of condition/action pairs, and executes each condition in turn until one is true, whereupon it executes the associated action. If the last condition has no action, then it acts as the else clause in the if. For example:

```
(if {~ $#var 0} {
        echo zero elements
    }
    {~ $#var 1} {
        echo one element
    }
    {echo more than one element}
)
```

Std provides various other control structures. And and or provide the equivalent of *rc*'s && and || operators. They each take any number of command block arguments and conditionally execute each in turn. And stops executing when a block's status is false, or when a block's status is true:

```
and {~ $var 1} {~ $var '*.sbl'} {echo variable ends in .sbl}
(or {mount /dev/eia0 /n/remote}
    {echo mount has failed with $status}
)
```

An extremely easy trap to fall into is to use $* inside a block assuming that its value is the same as that outside the block. For instance:

```
# this will not work
if {~ $#* 2} {echo two arguments}
```

It will not work because $* is set locally for every block, whether it is given arguments or not. A solution is to assign $* to a variable at the start of the block:

```
args = $*
if {~ $#args 2} {echo two arguments}
```

While provides looping, executing its second argument as long as the status of the first remains true. As the status of an empty block is always true,

```
while {} {echo yes}
```

will loop forever printing yes. Another looping command is getlines, which loops reading lines from its standard input, and executing its command argument, setting the environment variable $line to each line in turn. For example:

```
getlines {
    echo '#' $line
} < x.b
```

will print each line of the file x.b preceded by a # character.

Exceptions

When the shell encounters some error conditions, such as a parsing error, or a redirection failure, it prints a message to standard error and raises an *exception*. In an interactive shell this is caught by the interactive command loop; in a script it will cause an exit with a false status, unless handled.

Exceptions can be handled and raised with the rescue and raise commands provided by std. An exception has a short string associated with it.

```
raise error
```

will raise an exception named error.

```
rescue error {echo an error has occurred} {
    command
}
```

will execute command and will, in the event that it raises an error exception, print a diagnostic message. The name of the exception given to rescue can end in an asterisk (*), which will match any exception starting with the preceding characters. The * needs quoting to avoid being expanded as a wildcard by the shell.

```
rescue '*' {echo caught an exception $exception} {
     command
}
```

will catch all exceptions raised by command, regardless of name. Within the handler block, rescue sets the environment variable $exception to the actual name of the exception caught.

Exceptions can be caught only within a single process — if an exception is not caught, then the name of the exception becomes the exit status of the process. As *sh* starts a new process for commands with redirected I/O, this means that

```
raise error
echo got here
```

behaves differently to:

```
raise error > /dev/null
echo got here
```

The former prints nothing, while the latter prints got here.

The exceptions break and continue are recognised by std's looping commands for, while, and getlines. A break exception causes the loop to terminate; a continue exception causes the loop to continue as before. For example:

```
for i in * {
     if {~ $i 'r*'} {
          echo found $i
          break
     }
}
```

will print the name of the first file beginning with r in the current directory.

Substitution builtins

In addition to normal commands, a loaded module can also define *substitution builtin* commands. These are different from normal commands in that they are executed as part of the argument gathering process of a command, and instead of returning an exit status, they yield a list of values to be used as arguments to a command. They can be thought of as a kind of active environment variable, whose value is created every time it is referenced. For example, the split substitution builtin defined by std splits up a single argument into strings separated by characters in its first argument:

```
echo ${split e 'hello there'}
```

will print

```
h llo th r
```

Note that, unlike the conventional shell backquote operator, the result of the $ command is not re-interpreted, for example:

```
for i in ${split e 'hello there'} {
     echo arg $i
}
```

will print

```
arg h
arg llo th
arg r
```

Substitution builtins can only be named as the initial command inside a dollar-referenced command block - they live in a different namespace from that of normal commands. For instance, loaded and ${loaded} are quite distinct: the former prints a list of all builtin names and their defining modules, whereas the former yields a list of all the currently loaded modules.

Std provides a number of useful commands in the form of substitution builtins. ${join} is the complement of ${split} : it joins together any elements in its argument list using its first argument as the separator, for example:

```
echo ${join . file tar gz}
```

will print:

```
file.tar.gz
```

The in-built shell operator $" is exactly equivalent to ${join} with a space as its first argument.

List indexing is provided with ${index} , which given a numeric index and a list yields the *index* th item in the list (origin 1). For example:

```
echo ${index 4 one two three four five}
```

will print

```
four
```

A pair of substitution builtins with some of the most interesting uses are defined by the shell itself. ${quote} packages its argument list into a single string in such a way that it can be later parsed by the shell and turned back into the same list. This entails quoting any items in the list that contain shell metacharacters, such as ; or & . For example:

```
x='a;' 'b' 'c d' ''
echo $x
echo ${quote $x}
```

will print

```
a; b c d
'a;' b 'c d' ''
```

Travel in the reverse direction is possible using ${unquote} , which takes a single string, as produced by ${quote} , and produces the original list again. There are situations in *sh* where only a single string can be used, but it is useful to be able to pass around the values of arbitrary *sh* variables in this form; ${quote} and ${unquote} between them make this possible. For instance the value of a *sh* list can be stored in a file and later retrieved without loss. They are also useful to implement various types of behaviour involving automatically constructed shell scripts; see Lexical binding, below, for an example.

Two more list manipulation commands provided by std are ${hd} and ${tl} , which mirror their Limbo namesakes: ${hd} returns the first element of a list, ${tl} returns all but the first element of a list. For example:

```
x=one two three four
echo ${hd $x}
echo ${tl $x}
```

will print:

```
one
two three four
```

Unlike their Limbo counterparts, they do not complain if their argument list is not long enough; they just yield a null list.

Std provides three other substitution builtins of note. ${pid} yields the process id of the current process. ${pipe} provides a somewhat more cumbersome equivalent of the >{} and <{} commands found in *rc*, i.e. branching pipelines. For example:

```
cmp ${pipe from {old}} ${pipe from {new}}
```

will regression-test a new version of a command. Using ${pipe} yields the name of a file in the namespace which is a pipe to its argument command.

The substitution builtin ${parse} is used to check shell syntax without actually executing a command. The command:

```
x=${parse '{echo hello, world}'}
```

will return a parsed version of the string echo hello, world ; if an error occurs, then a parse error exception will be raised.

Functions

Shell functions are a facility provided by the std shell module; they associate a command name with some code to execute when that command is named.

```
fn hello {
    echo hello, world
}
```

defines a new command, hello , that prints a message when executed. The command is passed arguments in the usual way, for example:

```
fn removems {
    for i in $* {
        if {grep -s Microsoft $i} {
            rm $i
        }
    }
}
removems *
```

will remove all files in the current directory that contain the string Microsoft.

The status command provides a way to return an arbitrary status from a function. It takes a single argument its exit status is the value of that argument. For instance:

```
fn false {
    status false
}
fn true {
    status ''
}
```

It is also possible to define new substitution builtins with the command `subfn`: the value of `$result` at the end of the execution of the command gives the value yielded. For example:

```
subfn backwards {
     for i in $* {
          result=$i $result
     }
}
echo ${backwards a b c 'd e'}
```

will reverse a list, producing:

```
d e c b a
```

The commands associated with shell functions are stored as normal environment variables, and so are exported to external commands in the usual way. `Fn` definitions are stored in environment variables starting `fn-`; `subfn` definitions use environment variables starting `sfn-`. It is useful to know this, as the shell core knows nothing of these functions – they look just like builtin commands defined by `std`; looking at the current definition of `$fn-`*name* is the only way of finding out the body of code associated with function *name*.

Other loadable *sh* modules

In addition to `std`, and `tk`, which is mentioned later, there are several loadable *sh* modules that extend *sh's* functionality.

`Expr` provides a very simple stack-based calculator, giving simple arithmetic capability to the shell. For example:

```
load expr
echo ${expr 3 2 1 + x}
```

will print `9`.

`String` provides shell level access to the Limbo string library routines. For example:

```
load string
echo ${tolower 'Hello, WORLD'}
```

will print

```
hello, world
```

`Regex` provides regular expression matching and substitution operations. For instance:

```
load regex
if {! match '^[a-z0-9_]+$' $line} {
     echo line contains invalid characters
}
```

`File2chan` provides a way for a shell script to create a file in the namespace with properties under its control. For instance:

```
load file2chan
(file2chan /chan/myfile
     {echo read request from /chan/myfile}
     {echo write request to /chan/myfile}
)
```

`Arg` provides support for the parsing of standard Unix-style options.

Sh and Inferno devices

Devices under Inferno are implemented as files, and usually device interaction consists of simple strings written or read from the device files. This is a happy coincidence, as the two things that *sh* does best are file manipulation and string manipulation. This means that *sh* scripts can exploit the power of direct access to devices without the need to write more long winded Limbo programs. You do not get the type checking that Limbo gives you, and it is not quick, but for knocking up quick prototypes, or wrapper scripts, it can be very useful.

Consider the way that Inferno implements network access, for example. A file called /net/cs implements DNS address translation. A string such as tcp!www.vitanuova.com!telnet is written to /net/cs ; the translated form of the address is then read back, in the form of a (*file* , *text*) pair, where *file* is the name of a *clone* file in the /net directory (e.g. /net/tcp/clone), and *text* is a translated address as understood by the relevant network (e.g. 194.217.172.25!23). We can write a shell function that performs this translation, returning a triple (*directory* *clonefile* *text*):

```
subfn cs {
     addr := $1
     or {
          <> /net/cs {
               (if {echo -n $addr >[1=0]} {
                         (clone addr) := `{read 8192 0}
                         netdir := ${dirname $clone}
                         result=$netdir $clone $addr
                    } {
                         echo 'cs: cannot translate "' ^
                              $addr ^
                              '":' $status >[1=2]
                         status failed
                    }
               )
          }
     } {raise 'cs failed'}
}
```

The code

```
<> /net/cs { ....}
```

opens /net/cs for reading and writing, on the standard input; the code inside the braces can then read and write it. If the address translation fails, an error will be generated on the write, so the echo will fail - this is detected, and an appropriate exit status set. Being a substitution function, the only way that cs can indicate an error is by raising an exception, but exceptions do not propagate across processes (a new process is created as a result of the redirection), hence the need for the status check and the raised exception on failure.

The external program read is invoked to make a single read of the result from /lib/cs . It takes a block size, and a read offset - it is important to set this, as the initial write of the address to /lib/cs will have advanced the file offset, and we will miss a chunk of the returned address if we re not careful.

Dirname is a little shell function that uses one of the *string* builtin functions to get the directory name from the pathname of the *clone* file. It looks like:

```
load string
subfn dirname {
    result = ${hd ${splitr $1 /}}
}
```

Now we have an address translation function, we can access the network interface directly. There are three main operations possible with Inferno network devices: connecting to a remote address, announcing the availability of a local dial-in address, and listening for an incoming connection on a previously announced address. They are accessed in similar ways (see *ip* (3) for details):

The dial and announce operations require a new net directory, which is created by reading the clone file – this actually opens the ctl file in a newly created net directory, representing one end of a network connection. Reading a ctl file yields the name of the new directory; this enables an application to find the associated data file; reads and writes to this file go to the other end of the network connection. The listen operation is similar, but the new net directory is created by reading from an existing directory's listen file.

Here is a *sh* function that implements some behaviour common to all three operations:

```
fn newnetcon {
    (netdir constr datacmd) := $*
    id := "{read 20 0}
    or {~ $constr ''} {echo -n $constr >[1=0]} {
        echo cannot $constr >[1=2]
        raise failed
    }
    net := $netdir/^$id
    $datacmd <> $net^/data
}
```

It takes the name of a network protocol directory (e.g. /net/tcp), a possibly empty string to write into the control file when the new directory id has been read, and a command to be executed connected to the newly opened data file. The code is fairly straightforward: read the name of a new directory from standard input (we are assuming that the caller of newnetcon sets up the standard input correctly); then write the configuration string (if it is not empty), raising an error if the write failed; then run the command, attached to the data file.

We set up the $net environment variable so that the running command knows its network context, and can access other files in the directory (the local and remote files, for example). Given newnetcon , the implementation of dial , announce , and listen is quite easy:

```
fn announce {
    (addr cmd) := $*
    (netdir clone addr) := ${cs $addr}
    newnetcon $netdir 'announce '^$addr $cmd <> $clone
}

fn dial {
    (addr cmd) := $*
    (netdir clone addr) := ${cs $addr}
    newnetcon $netdir 'connect '^$addr $cmd <> $clone
}

fn listen {
    newnetcon ${dirname $net} '' $1 <> $net/listen
}
```

Dial and announce differ only in the string that is written to the control file; listen assumes it is being called in the context of an announce command, so can use the value of $net to open the listen file to wait for incoming connections.

The upshot of these function definitions is that we can make connections to, and announce, services on the network. The code for a simple client might look like:

```
dial tcp!somewhere.com!5432 {
    echo connected to `{cat $net/remote}
    echo hello somewhere >[1=0]
}
```

A server might look like:

```
announce tcp!somewhere.com!5432 {
    listen {
        echo got connection from `{cat $net/remote}
        cat
    }
}
```

Sh and the windowing environment

The main interface to the Inferno graphics and windowing system is a textual one, based on Osterhaut's Tk, where commands to manipulate the graphics inside windows are strings using a uniform syntax not a million miles away from the syntax of *sh*. (See section 9 of Volume 1 for details). The tk *sh* module provides an interface to the Tk graphics subsystem, providing not only graphics capabilities, but also the channel communication on which Inferno's Tk event mechanism is based.

The Tk module gives each window a unique numeric id which is used to control that window.

```
load tk
wid := ${tk window 'My window'}
```

loads the tk module, creates a new window titled My window and assigns its unique identifier to the variable $wid . Commands of the form tk $wid *tkcommand* can then be used to control graphics in the window. When writing tk applets, it is helpful to get feedback on errors that occur as tk commands are executed, so here's a function that checks for errors, and minimises the syntactic overhead of sending a Tk command:

```
fn x {
    args := $*
    or {tk $wid $args} {
        echo error on tk cmd $"args':' $status
    }
}
```

It assumes that $wid has already been set. Using x , we could create a button in our new window:

```
x button .b -text {A button}
x pack .b -side top
x update
```

Note that the nice coincidence of the quoting rules of *sh* and tk mean that the unquoted *sh* command block argument to the button command gets through to tk unchanged, there to become quoted text.

Once we ve got a button, we want to know when it has been pressed. Inferno Tk sends events through Limbo channels, so the Tk module provides access to simple string channels. A channel is created with the chan command.

```
chan event
```

creates a channel named event . A send command takes a string to send down the channel, and the ${recv} builtin yields a received value. Both operations block until the transfer of data can proceed as with Limbo channels, the operation is synchronous. For example:

```
send event 'hello, world' &
echo ${recv event}
```

will print hello, world . Note that the send and receive operations must execute in different processes, hence the use of the & backgrounding operator. Although for implementation reasons they are part of the Tk module, these channel operations are potentially useful in non-graphical scripts they will still work fine if there s no graphics context.

The tk namechan command makes a channel known to Tk.

```
tk namechan $wid event
```

Then we can get events from Tk:

```
x .b configure -command {send event buttonpressed}
while {} {echo ${recv event}} &
```

This starts a background process that prints a message each time the button is pressed. Interaction with the window manager is handled in a similar way. When a window is created, it is automatically associated with a channel of the same name as the window id. Strings arriving on this are window manager events, such as resize and move . These can be interpreted if desired, or forwarded back to the window manager for default handling with tk winctl . The following is a useful idiom that does all the usual event handling on a window:

```
while {} {tk winctl $wid ${recv $wid}} &
```

One thing worth knowing is that the default exit action (i.e. when the user closes the window) is to kill all processes in the current process group, so in a script that creates windows, it is usual to fork the process group with pctl newgrp early on, otherwise it can end up killing the shell window that spawned it.

An example

By way of an example, I ll present a function that implements a simple network chat facility, allowing two people on the network to send text messages to one another, making use of the network functions described earlier.

The core is a function called chat which assumes that its standard input has been directed to an active network connection; it creates a window containing an entry widget and a text widget. Any text entered into the entry widget is sent to the other end of the connection; lines of text arriving from the network are appended to the text widget.

The first part of the function creates the window, forks the process group, runs the window controller and creates the widgets inside the window:

```
fn chat {
    load tk
    pctl newpgrp
    wid := ${tk window 'Chat'}
    nl := '
'   # newline
    while {} {tk winctl $wid ${recv $wid}} &
    x entry .e
    x frame .f
    x scrollbar .f.s -orient vertical -command {.f.t yview}
    x text .f.t -yscrollcommand {.f.s set}
    x pack .f.s -side left -fill y
    x pack .f.t -side top -fill both -expand 1
    x pack .f -side top -fill both -expand 1
    x pack .e -side top -fill x
    x pack propagate . 0
    x bind .e '<Key-'^$nl^'>' {send event enter}
    x update
    chan event
    tk namechan $wid event event
```

The middle part of `chat` loops in the background getting text entered by the user and sending it across the network (also putting a copy in the local text widget so that you can see what you have sent.

```
    while {} {
        {} ${recv event}
        txt := ${tk $wid .e get}
        echo $txt >[1=0]
        x .f.t insert end '''me: '^$txt^$nl
        x .e delete 0 end
        x .f.t see end
        x update
    } &
```

Note the null command on the second line, used to wait for the receive event without having to deal with the value (there's only one event that can arrive on the channel, and we know what it is).

The final piece of `chat` gets lines from the network and puts them in the text widget. The loop will terminate when the connection is dropped by the other party, whereupon the window closes and the chat finished:

```
    getlines {
        x .f.t insert end '''you: '^$line^$nl
        x .f.t see end
        x update
    }
    tk winctl $wid exit
}
```

Now we can wrap up the network functions and the chat function in a shell script, to finish off the little demo:

```
#!/dis/sh
Include the earlier function definitions here.
fn usage {
	echo 'usage: chat [-s] address' >[1=2]
	raise usage
}

args=$*
or {~ $#args 1 2} {usage}
(addr args) := $*
if {~ $addr -s} {
	# server
	or {~ $#args 1} {usage}
	(addr nil) := $args
	announce $addr {
		echo announced on `{cat $net/local}
		while {} {
			net := $net
			listen {
				echo got connection from `{cat $net/remote}
				chat &
			}
		}
	}
} {
	or {~ $#args 0} {usage}
	# client
	dial $addr {
		echo made connection
		chat
	}
}
```

If this is placed in an executable script file named chat, then

```
chat -s tcp!mymachine.com!5432
```

would announce a chat server using tcp on mymachine.com (the local machine) on port 5432.

```
chat tcp!mymachine.com!5432
```

would make a connection to the previous server; they would both pop up windows and allow text to be typed in from either end.

Lexical binding

One potential problem with all this passing around of fragments of shell script is the scope of names. This piece of code:

```
fn runit {x := Two; $*}
x := One
runit {echo $x}
```

will print Two, which is quite likely to confound the expectations of the person writing the script if they did not know that runit set the value of $x before calling its argument script. Some functional languages (and the *es* shell) implement *lexical binding* to get around this problem. The idea is to derive a new script from the old one with all the necessary variables bound to their current values, regardless of the context in which the script is later called.

Sh does not provide any explicit support for this operation; however it is possible to fake up a reasonably passable job. Recall that blocks can be treated as strings if necessary, and that ${quote} allows the bundling of lists in such a way that they can later be extracted again without loss. These two features allow the writing of the following let function (I have omitted argument checking code here and in later code for the sake of brevity):

```
subfn let {
	# usage: let cmd var...
	(let_cmd let_vars) := $*
	if {~ $#let_cmd 0} {
		echo 'usage: let {cmd} var...' >[1=2]
		raise usage
	}
	let_prefix := ''
	for let_i in $let_vars {
		let_prefix = $let_prefix ^
			${quote $let_i}^':='^${quote $$let_i}^';'
	}
	result=${parse '{'^$let_prefix^$let_cmd^' $*}'}
}
```

Let takes a block of code, and the names of environment variables to bind onto it; it returns the resulting new block of code. For example:

```
fn runit {x := hello, world; $*}
x := a 'b c d' 'e'
runit ${let {echo $x} x}
```

will print:

```
a b c d e
```

Looking at the code it produces is perhaps more enlightening than examining the function definition:

```
x=a 'b c d' 'e'
echo ${let {echo $x} x}
```

produces

```
{x:=a 'b c d' e;{echo $x} $*}
```

Let has bundled up the values of the two bound variables, stuck them onto the beginning of the code block and surrounded the whole thing in braces. It makes sure that it has valid syntax by using ${parse}, and it ensures that the correct arguments are passed to the script by passing it $*.

Note that all the variable names used inside the body of let are prefixed with let_. This is to try to reduce the likelihood that someone will want to lexically bind to a variable of a name used inside let.

The module interface

It is not within the scope of this paper to discuss in detail the public module interface to the shell, but it is probably worth mentioning some of the other benefits that *sh* derives from living within Inferno.

Unlike shells in conventional systems, where the shell is a standalone program, accessible only through exec(), in Inferno, *sh* presents a module interface that allows programs to gain lower level access to the primitives provided by the shell. For example, Inferno programs can make use of the shell syntax parsing directly, so a shell command in a configuration script might be checked for correctness before running it, or parsed to avoid parsing overhead when running a shell command within a loop.

More importantly, as long as it implements a superset of the Shell builtin interface, an application can load *itself* into the shell as a module, and define builtin commands that directly access functionality that it can provide.

This can, with minimum effort, provide an application with a programmable interface to its primitives. I have modified the Inferno window manager wm, for example, so that instead of using a custom, fairly limited format file, its configuration file is just a shell script. Wm loads itself into the shell, defines a new builtin command menu to create items in its main menu, and runs a shell script. The shell script has the freedom to customise menu entries dynamically, to run arbitrary programs, and even to publicise this interface to wm by creating a file with file2chan and interpreting writes to the file as calls to the menu command:

```
file2chan /chan/wmmenu {} {menu ${unquote ${rget data}}}
```

A corresponding wmmenu shell function might be written to provide access to the functionality:

```
fn wmmenu {
     echo ${quote $*} > /chan/wmmenu
}
```

Inferno has blurred the boundaries between application and library and *sh* exploits this the possibilities have only just begun to be explored.

Discussion

Although it is a newly written shell, the use of tried and tested semantics means that most of the normal shell functionality works quite smoothly. The separation between normal commands and substitution builtins is arguable, but I think justifiable. The distinction between the two classes of command means that there is less awkwardness in the transition between ordinary commands and internally implemented commands: both return the same kind of thing. A normal command s return value remains essentially a simple true/false status, whereas the new substitution builtins are returning a list with no real distinction between true and false.

I believe that the decision to keep as much functionality as possible out of the core shell has paid off. Allowing command blocks as values enables external modules to define new control-flow primitives, which in turn means that the core shell can be kept reasonably static, while the design of the shell modules evolves independently. There is a syntactic price to pay for this generality, but I think it is worth it!

There are some aspects to the design that I do not find entirely satisfactory. It is strange, given the throwaway and non-explicit use of subprocesses in the shell, that exceptions do not propagate between processes. The model is Limbo s, but I am not sure it works perfectly for *sh*. I feel there should probably be some difference between:

```
raise error > /dev/null
```

and

```
status error > /dev/null
```

The shared nature of loaded modules can cause problems; unlike environment variables, which are copied for asynchronously running processes, the module instances for an asynchronously running process remain the same. This means that a module such as tk must maintain mutual exclusion locks to protect access to its data structures. This could be solved if Limbo had some kind of polymorphic type that enabled the shell to hold some data on a module s behalf it could ask the module to copy it when necessary.

One thing that is lost going from Limbo to *sh* when using the tk module is the usual reference-counted garbage collection of windows. Because a shell-script holds not a direct handle on the window, but only a string that indirectly refers to a handle held inside the tk module, there is no way for the system to know when the window is no longer referred to, so, as long as a tk module is loaded, its windows must be explicitly deleted.

The names defined by loaded modules will become an issue if loaded modules proliferate. It is not easy to ensure that a command that you are executing is defined by the module you think it is, given name clashes between modules. I have been considering some kind of scheme that would allow discrimination between modules, but for the moment, the point is moot there are no module name clashes, and I hope that that will remain the case.

Credits

Sh is almost entirely an amalgam of other peoples ideas that I have been fortunate enough to encounter over the years. I hope they will forgive me for the corruption I ve applied...

I have been a happy user of a version of Tom Duffs *rc* for ten years or so; without *rc*, this shell would not exist in anything like its present form. Thanks, Tom.

It was Byron Rakitzis UNIX version of *rc* that I was using for most of those ten years; it was his version of the grammar that eventually became *sh* s grammar, and the name of my glom() function came straight from his *rc* source.

From Paul Haahr s *es*, a descendent of Byrons *rc*, and the shell that probably holds the most in common with *sh*, I stole the blocks as values idea; the way that blocks transform into strings and vice versa is completely *es* s. The syntax of the if command also comes directly from *es*.

From Bruce Ellis *mash*, the other programmable shell for Inferno, I took the load command, the "{} syntax and the <> redirection operator.

Last, but by no means least, S.R. Bourne, the author of the original *sh*, the granddaddy of this *sh*, is indirectly responsible for all these shells. That so much has remained unchanged from then is a testament to the power of his original vision.