TkPerl— A port of the Tk toolkit to Perl5

Malcolm Beattie

Oxford University Computing Services

mbeattie@sable.ox.ac.uk

ABSTRACT: TkPerl is a port (work in progress) of the Tk[1] toolkit to Perl5[2]. It takes advantage of Perl5's object oriented features and magic variables to implement the Tk toolkit in Perl5. Nothing passes through the Tcl parser so knowledge of Tcl syntax is not required to use TkPerl. TkPerl is freeware (distributed under the GNU General Public License) and is currently in alpha testing. Section 1 of the paper introduces TkPerl and is followed by a brief section on its target uses.. Since TkPerl relies heavily on the object oriented features of Perl5 (which is itself only just into beta test), section 3 explains how Perl5 implements classes, objects and methods. Section 4 discusses the differences between Tk/Tcl and TkPerl both at scripting level and at C level. Section 5 looks at some of the porting issues and problems and how Tcl conventions affect the design of Tk itself. Section 6 explains the current intentions for the future of TkPerl and section 7 gives availability information.

1. Introduction

TkPerl started life in November 1993 as an exercise in looking at the hooks Perl5 provided for interfacing to external C functions. I wanted to write a graphical front end to a network client program and for various reasons I wanted to write it in Perl. I was familiar with Tcl and Tk and took a close look at the internals of Tk. Much to my surprise, the code had few dependencies on Tcl parsing and commands (as opposed to Tcl tools such as the hash table functions). I decided to try to "port" Tk to Perl by replacing the Tcl-dependent parts of Tk with Perl equivalents. By that, I mean removing the Tcl-as-scripting-language parts of Tk (not the Tcl-as-C-toolkit parts) and replacing them with a Perl-ish interface. Most importantly,

- callbacks to Perl were to happen directly, without passing through the Tcl parser;
- Perl variables were to be traced for widgets like checkbuttons and radiobuttons so that the widgets could update themselves when required; and
- the Perl interface to the TkPerl toolkit should make use of the object oriented features of Perl5— classes, objects and methods.

A more irritating obstruction to a clean port than those mentioned above became evident as work progressed— I will discuss this later.

TkPerl is being developed under OSF/1 and Linux. It has also been tested under various versions of SunOS, Ultrix, Solaris, NeXT and a few other platforms too. It is released under the GNU General Public License. Since the first public release of TkPerl, there have been over 1000 downloads of it by anonymous ftp from its home site in the UK (as of 1 September 1994). There is also a mirror site in the US for which I do not have statistics.

The current (alpha4) release of TkPerl is in the form of an extension to Perl5. This means that it follows certain conventions for its building, loading and programmer interface. It can be built at the same time

[1] Tk/Tcl, written by John Ousterhout.

[2] Perl5, written by Larry Wall.

as Perl itself as a static or dynamically loadable extension or it can be built separately from Perl as a dynamically loadable extension. For dynamic loading to be possible, it must be supported by Perl on that platform. Currently, at least the following platforms are supported for this: OSF/1, SunOS, Solaris, Linux (via GNU dld) HPUX and NeXT. If the Tk extension is to be built at the same time as Perl, the TkPerl source distribution can simply be extracted and "Tk" appended to the list of extensions that Perl's Configure asks about. The Tcl include file directory may need to be specified beforehand, though. The programmer interface for this extension is to use the command "use Tk" near the top of the script. That causes the extension to be dynamically loaded where necessary, bootstrapped, initialised and to import the necessary symbols for subroutines and variables from the Tk package namespace. The Tk extension co-exists happily with other extensions, without any namespace pollution other than the documented class names and public routines such as **tkpack** or **addasyncio.**

2. Target uses for TkPerl

Rather than letting comparison between Tk+Tcl and Tk+Perl5 become a comparison between Tcl and Perl5, I'd like to suggest a somewhat different viewpoint for TkPerl. Many large applications are written in Perl and rely on Perl's native features (regular expressions, associative arrays, low-level system access) for power, Perl's compilation and optimisation for speed, Perl's warnings and perldb.pl for debugging and Perl's taint checking for security. Looked at in this way, Perl is comparable to C and TkPerl should be compared to the combination Tk/C rather than the combination Tk/Tcl. There is, as yet, no documentation on the C interface to Tk although I believe interest in this has been increasing recently. For example, TkPerl provides the asynchronous I/O facilities of Tk: these are available with the C interface to Tk but not the Tcl interface. TkPerl provides access to the Tcl interpreter used by Tk via a **tclcmd()** routine, although the widget command procedures are not available in there. A TkPerl script can use the Tcl interpreter in the same ways as a C program can: for allowing flexible user configuration and for allowing access to user-written C functions. I think that the latter is less likely to be relevant in TkPerl, though, since Perl5 provides similar functionality.

Having made the above suggestion, I should probably point out that, in practice, many potential users of TkPerl have seen it in a different light. I have been in touch with many perl users who are eager for access to the Tk toolkit but who do not wish to use Tcl for one reason or another.

3. A brief introduction to Perl5 objects and methods

Since TkPerl makes much use of the object oriented features of Perl5 and Perl5 itself is still only in beta test, I will explain a little bit about how classes, objects and methods are implemented in Perl5.

3.1. Packages

Those familiar with Perl4 will know that all subroutine names and variable names are qualified (even if only implicitly) by a package name. Variables can also be explicitly qualified by a package. The syntax for scalar variable **bar** in package Foo is $Foo'bar in Perl4 and $Foo::bar in Perl5. Variables which are not qualified by an explicit package are by default put in package "main". That default package can be changed by a command such as

```
package Foo;
```

which affects all variables from there to the end of the (lexically scoped) containing block.

3.2. References

In Perl4 variables are either scalars (strings or numbers), arrays (indexed by integers) or associative arrays (indexed by strings). Those fluent in Perl will realise that I haven't mentioned globs or filehandles but we will not concern ourselves with those here. In Perl5, there is another scalar type too. That type is a "reference" which can hold a reference to any other type (scalar, array or associative array). Syntactically, one prepends a backslash to any other variable to get a reference to that variable. Recall that we prefix identifiers with "$" for a scalar variable, "@" for an array or "%" for an associative array. Array constants can be comma-separated lists surrounded by parentheses. Associative array constants are of the same form but are taken in pairs of key followed by value.

Given the code

```
@evens = (2, 4, 6, 8);
$evenref = \@evens;
```

the scalar variable $evenref is a reference to the array @evens (and $evenref knows internally what type of variable it references). The syntactical rule in Perl5 for dereferencing references is as follows. Wherever it is legal to have a sequence of letters identifying a variable, it is also legal to have a reference to a variable of that type. For example, given the above code, @$evenref is the actual array @evens. The 0th element of the array can be referred to as either $evens[0] or $$evenref[0].

In Perl4, subroutines are defined with the syntax

```
sub foo {
    ...
}
```

and invoked with the syntax

```
&foo($arg1, $arg2, $arg3);
```

In Perl5 syntax, the "&" can be dropped when invoking subroutine foo and scalar variables can hold a reference to a subroutine. The code snippets

```
$fooref = \&foo;
&$fooref($arg1, $arg2, $arg3);
```

and

```
foo($arg1, $arg2, $arg3);
```

both invoke subroutine foo in the same way as the preceding example.

3.3. Blessing

Access to the object oriented features of Perl5 is by means of the simple but powerful **bless** command. This command takes as argument a scalar variable which contains a reference and “tags” that variable which the name of the current package (or with the package named by the optional second argument of the **bless** command). By “tags”, I mean that Perl remembers that that package is associated with the variable. By slight abuse of language, the reference is called a *blessed reference*. No variable’s value is affected, but the **ref** function applied to that reference will return the name of the package into which it is blessed (or the special out-of-band value **undef** if it is not blessed at all).

3.4. Classes and methods

How does the **bless** command make classes and methods possible? Well, suppose $obj is a scalar variable holding a reference blessed into package Foo. Leaving aside inheritance for a moment, Perl5 uses the syntax

```
$obj->somemeth($arg1, $arg2);
```

to mean the same as

```
&Foo::somemeth($obj, $arg1, $arg2);
```

In other words, Perl looks for subroutine &somemeth in package Foo and invokes it with $obj prepended to the other arguments passed. What if package Foo has no subroutine &somemeth? That is where inheritance comes in. The special array variable @ISA in package Foo is taken to be an (ordered) list of package names. If there is no subroutine &somemeth in package Foo, then Perl5 searches through all the packages named in @Foo::ISA for a subroutine &somemeth and executes the first one it comes to. If Perl *still* cannot find a subroutine &somemeth, then there are other fallbacks which allow autoloading and universal classes but I will not go into those here.

If you apply the -> operator with a bare word in place of $obj above, then it is taken to be a package name in which to start the subroutine search. The first argument passed to the subroutine it finds is that bare word. For example, assuming a subroutine &Foo::new exists, the code,

```
$foo = Foo->new($arg1, $arg2);
```

calls subroutine &new in package Foo with arguments ("Foo", $arg1, $arg2). The example above is rather a special one because, purely by convention, the subroutine &new in a package is the constructor for that package. In other words, it is expected to return a scalar variable containing a reference blessed into that package. The resulting scalar (assigned to $foo above) can be used in method calls such as

```
$foo->flash();
$foo->disable();
@textconf = $foo->configure("-text");
```

assuming that package Foo has defined appropriate subroutines &flash, &disable and &configure. A *class* in Perl5 is (by convention) just a package which has a constructor and whose subroutines are intended to be called as methods. An *object* in Perl5 is (again, by convention) just a blessed reference returned by a class. When a constructor for a class wants to return an object, it

- creates a reference to a variable (usually an array or an associative array) which contains any public or private data for that object;
- blesses the reference; and
- returns the reference.

When any of its subroutines (methods) are called, the subroutine dereferences its first argument to get at the object's data.

3.5. Syntactic sugar

There are some syntactic variations which are useful for common idioms. Method calls of the form

```
$obj->meth($some, $args);
$foo = Class->constr($some, $more, $args);
```

can also be written in the form

```
meth $obj ($some, $args);
$foo = constr Class ($some, $more, $args);
```

There is also syntax (which is rather more than mere sugar) which enables creation of references to anonymous arrays, anonymous associative arrays and anonymous subroutines. For example, the code

```
$listref = [1, 2, 3, [7, 8, 9]];
$hashref = { "key" => "value", "anotherkey" => "anothervalue" };
$callback = sub { print "Hello world\n" };
```

creates three references. Dereferencing them, we get

- @$listref, which is an array with four elements. The fourth element is a reference to an array with the three elements (7, 8, 9).
- %$hashref, which is an associative array with keys "key" and "anotherkey" whose respective values are "value" and "anothervalue". The symbol => is syntactic sugar for a plain comma but may be used in the future for something a bit cleverer.
- &$callback, a subroutine which prints a traditional greeting when invoked.

4. Differences between Tk/Tcl and TkPerl

This section outlines the differences between Tk/Tcl and TkPerl. The section splits fairly naturally into two parts, according to the user of TkPerl. There is, of course, no need to mention the end-user of a given Tk program in the comparison since all differences are at a lower level. I would think that the majority of TkPerl users would give their attention to the differences between TkPerl and Tk/Tcl as regards writing X clients purely at the script level. Any differences between additional non-X C functions that are bound in really fall outside the bounds of Tk-specific paper. However, writers of new widgets may well be interested in the differences between TkPerl widget code and Tk/Tcl widget code. For a simple widget, there is likely to be very little difference. Moreover, if any widget is written from scratch it can be written in such a way that minimal changes are necessary to go from a Tk/Tcl widget to a TkPerl widget.

4.1. Differences at scripting level

Provided the Tk extension is available to the perl executable (either statically linked into the perl executable or, more likely, dynamically loadable from a sharable object file), any perl script just needs the command

```
use Tk;
```

somewhere near its beginning to be able to make use of TkPerl. An average script using TkPerl would then do some initial processing followed by

```
$top = tkinit();
```

to initialise Tk and prepare a top level widget. Then it would create lots of widgets and finally call **tkmainloop** which invokes the main event loop. Callback are done to subroutines (possibly in their guise as methods), which can be defined anywhere, as usual in Perl. The **tkinit** command takes up to three arguments (all optional) to allow non-default application name, DISPLAY and X server synchronisation (this last for debugging).

4.1.1. Creating widgets

Whereas widgets in Tk/Tcl are created by a Tcl command whose name is that of the widget class (e.g. **button**), in TkPerl each widget class is a Perl class (e.g. Button), with a constructor method **new**. The **new** method takes as first argument either the parent widget object or else the Tk pathname you want to give the widget. (The return value of **tkinit** is a widget object referring to a toplevel widget. Tk/Tcl sets this widget up automatically at script start up and calls it "."). Later arguments are optional and are pairs of initial configuration requests.

Example

```
Tk/Tcl      button .b -text foo
TkPerl      $b = Button::new($top, "-text" => "foo");
```

$b above is a reference to the pathname, blessed into the appropriate widget class.

4.1.2. Calling widget methods

In Tk/Tcl creation of a widget causes creation of a new Tcl command whose name is the widget's pathname. Invoking this command with first argument **foo** invokes method **foo** on that widget. In TkPerl, the value returned from the constructor **new** of a widget class is a blessed reference. Widget methods are real methods in that class and so one can use the blessed reference to invoke them.

Example

```
Tk/Tcl      .b flash
            .b configure -textvariable bar
TkPerl      $b->flash();
            $b->configure("-textvariable" => $bar);
```

There are a few minor differences in names: delete, index and select are Perl keywords and have been renamed tkdelete, tkindex and tkselect. Tcl "submethods" such as

```
.some.widget scan mark
```

have been turned into single methods: scanmark, scandragto, selectclear, selectadjust, selectfrom, selectto. When a **configure** method is used to return information about a widget option TkPerl returns a standard Perl list with the same information returned by Tk/Tcl. When the **configure** method is used to return information about all options of a widget TkPerl returns a list each of whose elements is a reference to an information list about one particular option. For example, after executing

```
@conf = $w->configure();
```

$conf[0] is a reference to a list containing either five elements (for an ordinary option) or two elements (for an alias option).

4.1.3. Callbacks

Configuration options where Tk/Tcl has a callback command (e.g. **-command, -yscrollcommand**) use slaves and methods instead in TkPerl (**-slave**, **-method**, **-yscrollslave**, **-yscrollmethod**). A slave/method pair is intended to be a flexible way to define a callback which is either

- an ordinary subroutine together with a piece of clientdata; or
- an object together with a method to invoke on it.

Suppose you have a widget $w which you configure as follows:

```
$w->configure("-slave" => $slave, "-method" => $method);
```

Callback behaviour depends on whether $slave is a blessed reference or not. If $slave is a blessed reference then the method $method is invoked on the object $slave. If $method is "foo" then it is the same as doing

```
$slave->foo();
```

in pure Perl. If $method is blank then no callback happens. On the other hand, if $slave is not a reference (in particular, if $slave is not defined), then $method is taken to be an ordinary subroutine and is called with $slave as its first argument (or **undef** if $slave is not defined). $method itself can be either an ordinary variable which is taken to be the name of a subroutine or else it can be a subroutine reference. The subroutine reference can either be of the form \&foo or an anonymous subroutine sub { ... }. This latter form gives the TkPerl programmer a way of writing callback code right next to the action which should trigger it. Moreover, the callback code is still compiled by Perl just as any ordinary subroutine is. This gives better performance than **eval** or its equivalent, especially for callbacks such as mouse dragging callbacks triggered for motion events. Even if the name of a subroutine is given as a $method argument, the lookup from subroutine name to internal Perl code pointer happens only the first time the callback is triggered. Thereafter a cached code pointer is used, again making for better performance. In all cases, the callback routines are passed extra arguments in precisely the same situations as in Tk/Tcl (e.g. scrollbar callbacks).

4.1.4. Access to event fields

In Tk/Tcl event callbacks are specified as Tcl command strings in which magic cookies beginning with the percent character “%” are replaced with data from fields from the event associated with the callback. In TkPerl, no magic cookies are involved but the magic variables $Tk::EvW, $Tk::EvA and so on all refer dynamically to the appropriate data. One slight addition: since $Tk::EvW returns a widget pathname and sometimes blessed references are more useful, the subroutine &EvWref returns a blessed reference to the appropriate widget.

4.1.5. Variable tracking

In configuration options such as **-variable** for radiobuttons where Tk/Tcl expects the name of a Tcl variable (as a string), TkPerl expects a genuine perl variable.

Example

```
Tk/Tcl     .r configure -variable foo -value green
TkPerl     $r->configure("-variable" => $foo, "-value" => "green");
```

Any alteration to that variable is tracked by the widget and it updates itself as necessary.

4.1.6. Binding events

Instead of Tk/Tcl **bind**, TkPerl has a **tkbind** command, since **bind** is already a Perl keyword:

```
tkbind($w, "<SomeEventName>", \&someaction);
```

$w can be a blessed reference to a widget or a widget pathname or a string corresponding to a widget class (e.g. "Button") or "all". The third argument is the method or subroutine half of a slave/method pair, as described in a previous section. An optional fourth argument is the corresponding slave/clientdata.

4.1.7. Processing a filehandle asynchronously

There is no Tk/Tcl equivalent for this, although Tk offers the service as C routine. The TkPerl command

```
addasyncio(FILEHANDLE, $selectfor, $method);
```

(with an optional fourth argument, $slave) arranges for the method to be called whenever one of the "interesting" conditions named in the $selectfor argument happens on FILEHANDLE. An "interesting" condition here means one where the file descriptor underlying the Perl filehandle FILEHANDLE becomes readable, writable or has an exception occur on it. When $method is called, it is called with arguments that tell it which condition(s) actually happened. When the conditions include being readable, one of the arguments is the number of bytes available to be read with blocking (if the O/S makes this information available).

4.1.8. Widget objects and inheritance

As alluded to above in a number of places, there are different ways of referring to widgets. You can use Tk pathnames, but you lose the advantages of perl objects and this use may be deprecated in a future release. The return value of each widget constructor method **new** is a reference to a string (the new pathname, in fact), blessed into the widget's class. When the first argument to a widget **new** method is a widget object (i.e. not a direct string pathname), an automatically generated unique pathname of a child of that widget is used instead (cf. the Xt toolkit and the Athena widget set). One way of constructing widget objects not yet referred to is very useful for simplifying widget inheritance for user-written composite widgets. A (blessed) reference to an associative array is acceptable as a widget object. In this case, when a widget method is applied to it (e.g. via @ISA inheritance) it looks for a key of that widget's classname and takes the corresponding value. If the key is not present it looks for the key "Default." The value found is then decoded as a widget by trying all the previously mentioned interpretations. For example, given code of the form

```
package Scrolledlistbox;
@ISA = (Listbox, Scrollbar);
sub new {
    # create a new listbox $l
    # create a new scrollbar $s
    # link the two together
    bless {Listbox => $l, Scrollbar => $s};
}
```

and the assignment

```
$sl = Scrolledlistbox::new($parent);
```

in a TkPerl script, $sl will understand the methods of both the Listbox class and the Scrollbar class. Those methods will be applied to the appropriate widget ($l or $s). Certain TkPerl commands look for other special keys (e.g. **focus** looks for key Focus). Since the widget object is a reference to an associative array, any public or private data fields for the object can be held as values in that associative array.

4.2. Differences at widget-writing level

We have already seen how composite widgets can be implemented in Perl scripts by imaginative use of inheritance. This section concerns how widgets themselves are implemented in C. This is more closely related to the following section on porting issues than to the preceding section on Perl classes. The C programmer's interface for writing new widgets is still rather fluid, especially as regards constructors. A future version of TkPerl will probably have a base Widget class from which fundamental methods can be inherited. In particular, the **new** constructor and the **configure** method for most widgets could be inherited. The widget writer would just register a class record containing such things as

- the classname,
- the configspecs table, and
- a list of method names.

The methods themselves can be written in perl or in C. If they are to be written in C, the methods can be in the form of XS files, which make use of the xsubpp pre-processor distributed with Perl5.

4.2.1. Widget constructors and methods in C

The widgets written in C which are currently distributed with TkPerl are all ports of the original Tk/Tcl widgets. They do not make use of Perl's XS pre-processing format because they were ported before xsubpp was developed. When the command "use Tk" is issued in a script, the sharable object code for Tk is loaded (if not already statically linked into the perl executable) and Tk's bootstrap function is invoked. This calls an initialisation routine for each widget which passes its class structure to TkPerl's **init_widget** utility. The fields in the class structure include the classname, a function pointer for the widget's constructor, a function pointer for the widget's methods and an array of integers corresponding to the widget's methods. The constructor function for widget class Foo is a slight modification of what Tk/C would call **Tk_FooCmd** and what a Tk/Tcl programmer would create with

```
foo .some.widget.path
```

Each element of the array of integers is an enumeration constant which indexes into a global array of standard method names. The **init_widget** function binds the constructor function to the Perl routine **&new** in package Foo. For each entry in the array of method integers, it binds the corresponding method name to the function pointer for the widget's methods. When the function is invoked, it will have available the method index integer for the name which invoked it. That function pointer is to a modification of what Tk/Tcl calls the **Tk_FooWidgetCmd** function and what the Tk/Tcl programmer would call as the ".some.widget.path" command from the example above. In TkPerl this function consists of a large **switch** statement which implements each method as a **case** instead of parsing a string corresponding to the method name, as Tk/Tcl does.

TkPerl offers three extra configuration option types to implement callbacks and variable tracking. In Tk/Tcl, callbacks are strings interpreted as Tcl commands and tracked variables are strings interpreted as names of Tcl variables. In TkPerl, callback subroutines are compiled and variable symbol lookup happens at compile time so new configuration types are necessary. Instead of a TK_CONFIG_STRING entry in the configspecs table, tracked variables use TKP_CONFIG_VARIABLE, methods use TKP_CONFIG_METHOD and slaves use TKP_CONFIG_SLAVE. In the widget structure itself, a tracked variable is stored with C type **SV *** and a method/slave pair is stored in a **struct tkpcallback** which automatically handles caching of method lookups for optimising speed. Invoking a callback from application code is done by pushing any extra arguments onto the Perl stack (if necessary) and invoking

TkpCallback(callback, n). Here, **callback** is the relevant **struct tkpcallback** and **n** is the number of extra arguments to pass to the callback (over and above the slave/clientdata argument that is automatically passed). The possible conversion of strings to subroutine references and the caching of Perl's internal pointers to them is all handled transparently by TkpCallback.

4.2.2. Utility functions

TkPerl supplies utility functions in C for the widget writer in addition to TkpCallback and the configspecs types mentioned above. These cover such things as

- converting from an SV (the internal type for Perl variables) to a widget record or Tk window structure;
- setting up traces on Perl variables (the equivalent of Tcl_TraceVar); and
- generating a new widget given a pathname or parent widget object.

5. Porting issues

At first sight, there were only two areas which would require significant work before a minimal Perl version of Tk would run. Those were:

(1) Many Tk/Tcl widgets have resources such as **-command** to store text for callbacks which are later interpreted as Tcl command strings using **Tcl_Eval().**

(2) Many Tk/Tcl widgets use the variable tracing facilities of Tcl (such as **Tcl_TraceVar**()) to carry out a C function whenever a Tcl variable is written to. For example, a checkbutton widget traces the Tcl variable named by its **-variable** option and sets or resets its visual marker accordingly whenever that variable is modified.

Perl5 provides similar facilities for C programmers but with rather different interfaces. On top of this, the infrastructure of each Tk widget command procedure (the C function bound to a widget's pathname) would need to be rewritten. This would just involve changing a series of "if-then-else" string comparisons to a **switch** statement to handle the way Perl5 calls XSUBs (C functions bound to perl subroutines, called usersubs in Perl4).

There turned out to be a more serious problem with the Tk/Tcl code as far as porting it to Perl5 was concerned. This wasn't a localised problem but a more general one: the argument passing conventions used by Tk/Tcl. Not only the Tcl command procedures but also most of the internal Tk widget procedures used argc/argv style arguments. In other words the callers would convert any arguments in to an array of null terminated strings and the called function would parse out the appropriate data types again. This is highly suited to interfacing to Tcl since the interface between Tcl scripts and their corresponding C functions uses this model.

For Perl5, however, it is far from the best way of handling argument passing. Perl5 shields the programmer both at scripting level and at the C level from data-type problems. A Perl5 XSUB receives its arguments on a global stack, which can be treated just like an array of opaque pointers. Moreover, by writing source using the new XS language and its xsubpp pre-processor, one can avoid dealing with the stack or data-type conversions altogether. One need only name the C types of the variables one wants to be passed and the xsubpp pre-processor generates C code to handle the internal Perl stack and the necessary data conversions. All Perl variables are internally modelled as *scalar values* (SV's), a generic opaque structure which knows what sort of data it contains. There are documented macros for getting at that data and the Perl5 internals handle data-type issues automatically. Not only can they give you the contents of that variable as whatever data-type you want (for example string, integer or double) but any data-type conversion happens only where necessary. That is because an SV can hold the data in multiple formats at the same time. Furthermore, Perl5 supports using the SV as an lvalue and supports data-types such as lists, associative arrays and references to any data-type at all.

Carrying over the argument passing conventions of Tk/Tcl into TkPerl would have made for clumsy code. For example, suppose a numeric argument is passed to a Perl XSUB. (Recall that an XSUB is a perl subroutine that happens to call out to an external function). The Perl compiler will already have chosen its appropriate type (integer or double) when the script was parsed. The XSUB has immediate access to a C variable of type int or double. For a direct interface to the C functions which implement the Tk internals, that int or double would need converting to its string representation. Then it would be passed to a function whose first job would be to convert its string argument back into a numeric. This is unavoidable for C functions directly called by the Tcl parser because every piece of data in a Tcl script is a string and all arguments are passed into C functions as string arrays. However, it seems a crying shame to take this route in Perl, where all necessary conversions, optimisations and caching of values has already been done. It also seems rather a shame that there is not a stricter boundary between the two apparent "layers" of Tk/Tcl. That is,

- the upper layer consisting of C functions bound into the Tcl interpreter and called with argc/argv pairs, and
- the lower layer of functions which deal directly with widget fields and Tk's windows.

In particular, consider the C function bound to a typical widget pathname command in Tk/Tcl. It not only parses its argument strings (for example **get, invoke,** or **nearest**) but it also carries out most of the resulting "method" commands. Those auxiliary functions which it does make use of tend to be declared **static,** to accept strings as arguments and do their own parsing. It is not possible to write a separate TkPerl function which takes values of the required type and calls directly to the low level functions. One workaround is to copy the whole source file and modify the low level functions slightly to accept arguments of the correct type without parsing them from strings. Another workaround is to convert the arguments that Perl passes back into strings, dynamically create a new array of strings and then call into the low level routine. The latter workaround still needs the whole source file to be copied since the low level functions are declared **static** and so cannot be linked in from the standard Tk library statically, let alone dynamically.

6. Future plans

TkPerl is still alpha-code. It lacks

- one of the standard Tk widgets (the Text widget);
- some of the standard bindings (such as menu bars and accelerators);
- Perl-ish versions of some auxiliary commands; and
- the Tk/Tcl **send** command (which I intend to implement with a more general Perl implementation of remote proxy objects).
  Perl5 itself is still in beta-test. Although the programmer interface to its internal functions aren't changing as frequently as they used to do, updating TkPerl to make use of newly documented interfaces and conventions is still non-trivial.

In the short term, I will continue tracking releases of Perl5, stabilise the current set of widgets and complete the set supplied, provide more Perl-ish interfaces to the Tk support commands and complete Tk.pm, the TkPerl equivalent of tk.tcl, written in Perl rather than Tcl. In the short to medium term, a decision has to be made. Currently, TkPerl is fairly close in many ways to Tk/Tcl. Apart from having widgets as Perl5 classes and their methods as true methods (allowing widget inheritance) there are few other major differences. It would be possible to take further advantage of Perl's object oriented features and rewrite major parts of Tk to behave differently. For example,

(1) the event delivery system could be rewritten to propagate events using Perl5's method inheritance,

(2) the widget configuration system could be rewritten to use method chaining (like with Xt's **set_values** method) simplifying widget inheritance even more,

(3) the main widget procedures could be allowed to be real Perl methods. Ways of interfacing Perl5 directly to C libraries are being developed and one could imagine new widget code being written directly in perl.

(4) Perl's **tie** command opens up many possibilities for useful interfaces to widgets.

However, it could also be argued that any major divergence of TkPerl from Tk/Tcl may cause more harm than good. It would make folding in future changes to Tk/Tcl harder, or even impossible, and would mean that programmers would be less able to cross to and fro between writing Tk/Tcl code and TkPerl code. Should the goal of TkPerl be to provide non-Tcl speakers with an alternative way of using the popular Tk toolkit or should the goal be to provide a native Perl GUI based on Tk? My current feeling is that the latter goal is more worthwhile.

7. Availability

TkPerl is freeware distributed under the GNU General Public License. It is currently available for anonymous ftp from ftp.ox.ac.uk in directory `/src/ALPHA`. The source-only distribution of the current alpha release is available as file `tkperl5a4.tar.gz`, a tar archive compressed with GNU gzip. Distributions which include executables (perl executables with the Tk extension linked in or else sharable object files for dynamic loading) may also be made available for some platforms. The most appropriate forum for discussion of TkPerl is probably the newsgroup `comp.lang.perl`. I am happy to receive email about TkPerl but can provide no guaranteed support for it.

Although TkPerl has been developed in my own time, I have been doing much of the development on the general Unix service system of my employers, Oxford University Computing Services. I would like to thank them for their encouragement and for an attitude towards free software that makes this kind of development work possible.

Malcolm Beattie gained his BA in Mathematics at Oxford University in 1989 and continued there to work towards his DPhil in Algebraic Topology. He joined Oxford University Computing Services in 1992 as a systems programmer and gained his DPhil in 1993.