Language Embedding by Dual Compilation and State Mirroring

Paul Tarau
Dépt. d'Informatique
Université de Moncton
tarau@info.umoncton.ca

Bart Demoen
Dept. of Computer Science
Katholieke Universiteit Leuven
bimbart@cs.kuleuven.ac.be

Abstract

We give an abstract scheme for the embedding of two languages with strong meta-programming, parsing and structure manipulation capabilities. We provide a call-back mechanism requiring each language to *compile* its service requests to the other's syntax for processing through the other's *meta-programming* capabilities. *State mirroring* ensures that only differential information needs to be transmitted. We describe a practical implementation of these principles and our experience with the **BinProlog Tcl/Tk** interface. Compilation by each side, that targets the other's syntax and dynamic object manipulation capabilities, achieves in *less than two hundred* lines of code, a fully functional integration of the two systems. The environment is portable to any Prolog system to which it adds powerful graphic and transparent distributed network programming facilities. Our approach suggest that logic programming language-processors can be made to fit easily in a multi-paradigm programming environment with little programming effort by using their strong *language translation* and *meta-programming* capabilities.

Keywords: multi-paradigm programming, visual environments for logic programming languages, meta-programming, Prolog graphic interface.

1 Introduction

Integration of various programming paradigms is a necessity in modern programming environments. Monolithic language implementations tend to be replaced more and more by a combination of specialised *tools*. In particular, generic graphic processors can be connected to various languages which do not need their own graphic primitives anymore.

However, when there's a large semantic distance between the languages and they feature radically different computational models, a low-level interface is a

painful and not always rewarding programming task. It happens very often that when the interface is finished one of the sides of the interface just becomes obsolete because of rapid evolution. This is especially true with various windowing systems which are often machine/operating system dependent.

The aim of this paper is to propose an unusually quick interfacing technique which uses meta-programming capabilities on the two sides. The basic idea is to link the two languages through a standard client-server interface and have each of them drive the other as an interactive agent, by generating the appropriate code on the fly. By mirroring the state of the objects, only state changes have to be transmitted from one side to the other, so that the granularity of the interaction will not become a bottleneck. Moreover, if one side has the capacity to react to events, this property will be inherited with minimal programming effort by the other side.

This paper is motivated by our work on finding a simple and portable way to add a visual programming environment to Prolog systems.

The popular Tcl/Tk visual language by John Ousterhout [2] is basically a a composite of a shell-like interpreted programming language (Tcl) and a high-level Motif-style graphic package (Tk). Tcl is an untyped string-only language with strong meta-programming facilities (i.e. an *eval* primitive and dynamic procedure creation).

After describing a general language-processor interaction model we will report how it has been applied to an interface between BinProlog [3], and Tcl/Tk and how this interface has been ported to another Prolog system (Prolog by BIM) with minimal programming effort.

2 An abstract multi-language communication model

Let `L` and `R` be two languages, $t_{L,R}$ a translation function (compilation) from L to R, $t_{R,L}$ the reverse translation from `R` to `L`. Let e_L and respectively e_R be the *eval* operations of `L` and `R`. Let r_L, w_L, r_R and w_R denote the read and write operations of languages L and R.

An L_R-processor is a process with capabilities to

1. execute programs written in L
2. implement a translation function from L to R.

We say that two language processors L_R and R_L are *connected* if the following conditions are verified:

1. each *translate+write* operation on one side triggers exactly one *read+eval* operation on the other side,

2. one or more[1] *translate+write* operations can be triggered by each *eval* operation.

This can be formalized more precisely in terms of temporal modal operators.

Our *write* and *read* operations should not be confused with ordinary input-output primitives. More precisely they are abstracted from the subset of input-output operations which can be meaningfully translated and are therefore suitable as input for the other side's eval functions. Thus they can be seen as carrying messages from an L_R processor to an R_L processor.

We say that an L_R and an R_L processor are *fairly interacting* if

1. each write operation on one side will be eventually read on the other side
2. the behaviour of each side is observable on the other side in terms of a sequence of successive read and write operations

If for two L_R and R_L language processors are *connected* and their read and write operations are queued, then they are *fairly interacting*.

We will break this fully symmetric behaviour by defining a *slave* language-processor as one that passively waits for write operations from the other side and a *master* language-processor which reacts to a write operation by a corresponding read operation with its result passed to its *eval* operation. We suppose also that the user interacts only with the master. As only the slave waits blocked on read operations, the master is free to be 'multi-threaded' or event-driven.

3 The BinProlog to Tcl/Tk interface

As an instance of this general scheme, we have chosen to connect BinProlog as a 'slave process' to a Tcl/Tk *wish* shell enhanced with reactive capabilities given by the *addinput* facility [2]. A specialized BinProlog toplevel is launched from the Tcl/Tk shell as a background Unix process connected through a bidirectional pipe. The BinProlog process is able to generate and react to Tcl/Tk events.

The interface uses tcl7.3 with tk3.6 combined with addinput-3.6a. It consists of 91 lines of Prolog and 101 lines of (commented) Tcl code. Thanks to the strong meta-programming capabilities of both languages the interface boldly *compiles* messages from one side to evaluable representations on the other side and calls the appropriate *eval* operation. The interface is portable to other Prolog systems and requires no C-programming. The installation of a BinProlog slave on the 'event channel' handled by `addinput` is done as follows:

[1] The ability to trigger more than one *translate+write* is needed for instance when more than one task has to be initiated or more than on component of the state of one side has to be updated in a transaction.

[2] A modification of the Tcl/Tk event loop (freely available from ftp.neosoft.com) which enables it to react to the presence of new input by triggering the execution of a user specified procedure when input becomes available.

```
proc start_prolog {} {
#   Opening a bidirectional pipe to BinProlog
#   under control of the addinput facility
    set f [open "|bp -q5 server.bp" r+]
    addinput $f "bpInOut %% %E %F"
}
```

Output from the BinProlog slave `bp server.bp` activates a Tcl/Tk procedure which executes on a *new* Tk-interpreter without disturbing normal interaction with the *wish* shell.

Basically, performance is not affected by the parsing as this is kept minimal. Moreover in systems with a parser written in C (as Prolog by BIM or the Koen De Bosschere's ISO-parser based version of BinProlog) parsing is in itself fast enough not to dominate the interaction cost. Techniques as *state mirroring* which will be discussed in the next section are also used to minimize the communication overhead.

High-level primitives on the Tcl/Tk side ensure that output to Prolog are valid Prolog terms. On the Prolog side a special toplevel loop reads the Tcl/Tk message, and applies its eval operation (call/1) to it. To avoid spurious interaction due to syntax errors or unexpected messages on each side only lines having a reserved header (i.e. `call_prolog` and respectively `call_tcl`) are evaluated. However, to give the look and feel of interacting with a Prolog system, messages from Prolog which are not of the form `call_tcl {...}` will be printed out as such. This fits well with Tcl which is string oriented but would not fit well with a term-oriented language like Prolog without parser modifications.

3.1 State mirroring

State mirroring is a simple technique that consists of duplication by a given language processor of (relevant) state information of an interactively connected other language processor.

In the case of Prolog, we have chosen to avoid the representation of states by infinite forward loops for two reasons:

1. unexpected failure would make the system unreliable and hard to debug
2. unexpected delays induced by garbage collection would make a visual interface unpleasant to use

Fortunately, BinProlog's blackboard turned out to be the perfect match to mirror the state of various Tcl/Tk objects. A library emulating BinProlog's blackboard operations in terms of generic Prolog dynamic database operations has been used to achieve the same effect in the Prolog by BIM port of the interface.

3.2 Programming with the embedded environment

Using the BinProlog-Tcl/Tk interface is very simple. Let `<P>` be the name of the composite program. Suppose the user creates 2 files `<P>.tcl` and `<P>.pl`. The composite 'program' can be executed as follows:

```
$ wish
% source <P>.tcl
```

`<P>.tcl` is expected to have at its end something like:

```
start_prolog
p {compile(<P>)}
```

The user will get the usual Tcl command prompt to interact with either Tcl or Prolog. A command entered at the Tcl prompt normally 'goes to' Tcl except for the following:

```
start_prolog      -connects to a new Prolog process
halt_prolog       -terminates the Prolog process:

p {<PrologTerm>}  -sends a goal to Prolog for quiet evaluation

q {<PrologTerm>}  -sends a query to Prolog and gets back answers
```

After being activated by one of the previous (p or q) Tcl commands a Prolog goal may 'call back' to continue the dialog with Tcl/Tk using the following Tcl command on the Prolog side:

```
call_tcl/1    -sends to Tcl a command to be immediately
               evaluated in 'background', while the Tcl shell
               prompts and waits for Tcl or Prolog commands.
```

Output with `puts` (Tcl) or `write/1` (Prolog) goes as usual to the `stdout` of the Tcl shell.

3.3 A visual N-queens program

This program illustrates two simple-minded but useful features of the interface:

1. Prolog lists are sent to Tcl as Tcl-lists
2. Prolog simply pumps solutions one by one using their most natural representation through `call_tcl(display_queens(L))`.
3. synchronization is done as follows:
   - the user provides mouse events generating write operations

- the prolog process waits for input on a read operation until asked for another answer

Equivalently, the user can also control the Prolog program from Tcl's command line by typing in "p more" or "p done" messages.

The code on the Prolog side consists of a classic N-queens program, the generic prolog interface program server.pl and the clause:

```
qs(N):-
    queens(N,L),
    call_tcl(display_queens(L)),   % sends a display request
    write('type <p done> when finished <p more> otherwise'),nl,
    in(call_prolog(done)),!.       % waits for 'done' or 'more'
```

The code on the Tcl/Tk side consists of the generic server.tcl interface program, routines to set up the visual display and procedures like `show_queen`, `hide_queen` which are called by the main Tcl/Tk-side routine:

```
proc display_queens {qs} {
  global w count
  incr count
  hide_queens
  set l 0
  foreach q $qs {
    incr l
    show_queen $w [expr $l -1] [expr $q -1]
  }
}
```

This procedure simply counts the answers and updates the display when a new answer comes from Prolog, triggered by the presence of new input on the pipe.

3.4 Performance evaluation

Tcl/Tk is itself a glue language which links together large blocks of C-code. We have written a naive-reverse program and measured that it executes in Tcl between 20-50 times slower than in BinProlog. This definitely suggests that our technique, using relatively expensive parsing and compilation operations will not become itself an interaction bottleneck and therefore due to the much higher flexibility and almost instant portability between various Prolog systems this approach looks superior to a highly Prolog-specific C-level integration.

As BinProlog is a much faster symbolic processor than Tcl, we have decided to do most of the translation operations on the Prolog side. We have measured about 1500 BinProlog-Tcl/Tk exchanges per second on a Sparcstation 2 and 700-800 exchanges on a Sparcstation 1, both serving the same Tektronix X-terminal through a network.

This confirms that the interface is fast enough for complex real time visualisation tasks and pleasant to use in a average Unix environment.

3.5 Porting the interface to another Prolog

It took one hour to make the interface work with ProLog by BIM. The main change was to modify `start_prolog` on the Tcl side to start a ProLog by BIM process with:

```
set f [open "|bim server.pro" r+]
```

The ProLog by BIM process is initialized from a small new toplevel `server.pro`. The one hour work had more to deal with the differences between the two Prolog systems, than the interface itself. Moreover, code written for one Prolog system (like the `N-queens` program) runs without changes on the other, as far as it is written in portable Prolog. We think this compares favorably with a dedicated piecewise interface written in C for each Prolog graphic builtin in terms of programming effort, portability and learning curve for the user, and, as our performance evaluation has shown, with reasonable real-time interactivity.

4 Related work

The only other Prolog with Tcl/Tk interface we know of is that developed by Micha Meier at ECRC [1] as the standard Eclipse GUI, although ongoing work on SICStus Prolog may also lead to a such an interface in the future. Micha Meier's interface is tightly integrated (at C-level) and Eclipse specific (although a Sicstus port is also provided). The tighter integration is better (in principle) in the case of a very low granularity communication, although the programming effort to implement it is more considerable than ours. Our performance analysis shows that global costs are dominated by process switching, graphic manipulations and brute force processing in Prolog. Moreover, our interface, does not rely on a specific Tcl/Tk implementation and has no modification to the C-code neither on the Prolog emulator neither on the Tcl/Tk side. Because of this reasons, we believe that the simplicity and portability of our design is a clear advantage, especially in the context of rapidly evolving backward incompatible Tcl/Tk releases.

5 Future work

We plan to make the interface fully Prolog-independent and to write Prolog libraries to support various existing Tcl/Tk extensions and tools for distributed Tcl/Tk programming, automatic interface generators etc. A Tcl/Tk based generic Prolog toplevel would also be very helpful to give the full illusion of *being in Prolog* while benefiting of visual goodies of a Tcl/Tk shell.

6 Conclusion

We have outlined a generic programming framework for a seamless integration of two programming languages which have 'eval' capability and implemented an instance of the framework as a Prolog to TCL/Tk interface.

Our experiments show that support for metaprogramming tools in modern programming environment is important not only for the expressiveness of the language itself but also multi-paradigm language integration.

We have shown that with a reduced programming effort a powerful interface can be built between a Prolog system and a state-of-the art visual environment.

This suggests that embedding of a logic engine in a generic multi-language environment is competitive in functionality and performance with the traditional approach which advocates to hardwire in the Prolog system a set of language-specific extensions.

The code for the interface can be obtained by ftp from **clement.info.umoncton.ca** and has been integrated in the BinProlog 2.20 distribution.

7 Acknowledgements

Paul Tarau thanks for support from K.U.Leuven through Fellowship F/93/36, from NSERC (grant OGP0107411) and from the FESR of the Université de Moncton. Bart Demoen thanks the Belgian Ministry (DPWB) for support through project IT/IF/4. Special thanks go to the referees for their useful comments and suggestions.

References

[1] M. Meier. The Eclipse tcl/tk interface, 1993. Program, publicly available on INTERNET. (ftp.ecrc.de).

[2] J. Ousterhout. Tcl/tk, 1993. Program, publicly available on INTERNET (harbor.ecn.purdue.edu).

[3] P. Tarau. Language issues and programming techniques in BinProlog. In D. Sacca, editor, *Proceeding of the GULP'93 Conference*, Gizzeria Lido, Italy, June 1993.