Arbeitsbereich DBIS
Fachbereich Informatik
Universität Hamburg
Vogt-Kölln Straße 30
D-22527 Hamburg (FRG)

Tycoon

Title:

P-Quest: Installation and User Manual

Author: Florian Matthes

Identification: DBIS Tycoon Report 101-91

Status: Initial Version

Date: October 1991

Description: This document explains the installation and use of the *P-Quest* system (version 14) on Sun-4 hardware platforms

Related Documents:
Tycoon Library Manual (*P-Quest* Version) [?]
The *P-Quest* C-Call Facility [?]
The Quest Language and System (Tracking Draft) [?]

Contents

1 What is *P-Quest*?

P-Quest was developed at the University of Hamburg, Germany, and adds orthogonal persistence to the programming language Quest developed by Luca Cardelli at DEC SRC, Palo Alto, USA [?]. The implementation of *P-Quest* was carried out in the ESPRIT-II basic research project FIDE and utilizes the Napier persistent object store provided by the University of St. Andrews, Scotland [?].

P-Quest is an intermediate step in the Tycoon project, a long-term research programme to develop an open database programming environment exploiting state-of-the-art programming language technology for the construction of complete data-intensive application systems. Specifically, the current *P-Quest* version is shipped with an initial set of Tycoon libraries that provide uniform iteration abstractions over a wide range of generic bulk data structures in addition to a standard library supporting base types and (formatted) file input and output. These libraries and a prototypical *P-Quest* Open-Look user interface library are described in a separate document [?].

P-Quest is strongly and statically typed and has three levels of entities (i) values, (ii) types and operators, (iii) kinds. Types classify values, and kinds classify types and type operators. Kinds are needed because the type level of *P-Quest* is unusually rich.

Evaluation is deterministic, left-to-right and applicative-order (call by value). *P-Quest* has a strong functional flair but it also incorporates imperative features (e.g. assignments). It provides higher-order functions, loops, conditionals, exceptions and a limited form of dynamic binding through modules.

The type level provides existential and universal type quantification, inductively defined subtyping over all type constructors, user-defined type operators and recursive types. Type annotations are only required in signatures since types and kinds within bindings are automatically inferred by the compiler.

Kinds are used to control the instantiation of type variables. They are means to denote sets of types, e.g., subtypes of a given type or sets of (higher-order) type operators with the same signature.

The syntax of *P-Quest* makes heavy use of initial and final keywords (resembling Modula-2 and Modula-3) but has virtually no commas and semicolons. New infix and listfix operators can be declared freely, hovever, there is no overloading of identifiers and all infix operators have the same precedence and are right-associative.

The *P-Quest* system runs on Sun-4 hardware platforms under SunOS 4.03 or higher. *P-Quest* expressions and programs are statically parsed, scoped, type-checked and compiled into portable byte-code. *P-Quest* programs can be either entered and evaluated interactively or stored in pre-compiled form as interfaces and modules that are linked dynamically. The present release of the system is equipped with a C-call facility to dynamically bind and call operating system or user-defined C library code [?].

The *P-Quest* compiler, linker, user data, compiled programs, compiled interfaces and their associated type and dependency information are all kept in a uniform persistent stable store. However, *P-Quest* source code is maintained outside the stable store in operating system text files. In *P-Quest* there is no distinction between volatile and persistent data. All entities (interfaces, modules, values, functions, types, kinds ...) that are reachable from named objects in the user environment outlive a single program execution. Storage allocated for temporary objects or persistent objects that are no longer reachable is automatically reclaimed by the system. There is no explicit object deletion operation in *P-Quest*.

2 Installing *P-Quest*

Sofware Version:	14 (based on Quest version 12a of DEC SRC)
Required Hardware:	Sun 4
Required OS Version:	SunOS 4.03 or higher
Space Requirements:	7 MB in /usr/local/lib
	8.2 MB for each stable store

To read the *P-Quest* tape, *cd* to a directory where *P-Quest* is to be installed. It should be on a file system and in a directory that is accessible by all future users of *P-Quest*. Then type:

```
tar xf /dev/rst1
```

This creates a directory *quest.p-sparc* in the current directory that contains all *P-Quest* binaries and library files. To install *P-Quest*, first become super user

```
su
```

Then create a symbolic link from the directory */usr/local/lib* to the newly created directory:

```
ln -s ‘pwd‘/quest.p-sparc /usr/local/lib
cd quest.p-sparc
```

Now you have to decide how to make the *P-Quest* binaries accessible to future *P-Quest* users. On most systems, new binaries are installed in the directory */usr/local/bin*. In this case, use the command

```
Install.p-sparc /usr/local/bin
```

Alternatively, you can choose any other directory that is on the shell search path, e.g.,

```
Install.p-sparc /usr/bin
```

The script *Install.p-sparc* just creates symbolic links and changes file access modes and therefore requires virtually no extra disk space. The files of the distribution directory are listed in Appendix ??.

3 Formatting and Initializing a New Persistent Store

P-Quest programs run against a persistent store that contains the complete *P-Quest* environment (programs and data). The first step to develop a *P-Quest* application system is therefore usually to create a private persistent store (implemented as a file in the Unix file system) and initialize it with the interactive compiler environment.

To create a persistent store in the current directory (called "." according to the Unix naming conventions), the following command has to be used:

```
PQFormat . 1200 300
```

The second parameter specifies the initial (and maximum) size of the persistent store file measured in 8K pages, while the third parameters specifies the number of 8K pages that are to be reserved as shadow storage to implement database recovery. In the example above, a file *stablestore* of 9830400 bytes will be created in the current directory. (1200-300) * 8192 bytes = 7.372800 bytes are available for user-data and 2457600 bytes will be used as shadow storage. Additionally, a file named *lockfile* will be created in the current directory. It is used to enforce exclusive access to the persistent store.

The next step is to initialize the the store with predefined *P-Quest* objects (built-in exceptions etc.):

```
PQInit .
```

The following command loads the interactive *P-Quest* system (compiler, top level, linker) into the persistent store:

```
PQuest . NewQuest.qm
```

This operation may take a while since the file *NewQuest.qm* contains data in a portable format that needs to be converted into the persistent store format. After the compiler is successfully loaded into the persistent store, all modules and interfaces listed in the file *Library.qst* are automatically imported.

Finally, the *P-Quest* prompt ("–") appears. Enter the following *P-Quest* commands to preserve the current status of the store:

```
import store :Store;
store.stabilise();
```

To leave the *P-Quest* system, type CTRL-D ("^D").

The steps described in this section have to be executed only once for each persistent store.

4 Interactive Commands

Once the persistent store is initialized, it suffices to issue the command

```
PQuest
```

to restart the compiler environment. All modules and interfaces listed in the file *Library.qst* which are not already cached in the persistent store are imported.

When the *P-Quest* prompt ("–") appears, you can evaluate expressions that have to be terminated by a semicolon,

```
3 + 4;
```

bind values, types and kinds to names,

```
let x = 3;
Let T = Int;
DEF SUBT = POWER(Int);
```

import compiled interfaces and modules,

```
import print :Print;
```

call routines from imported modules,

```
print.string("Hello World!\n");
```

read *P-Quest* source programs from files,

```
load"Library.qst";
```

compile *P-Quest* interfaces,

```
interface A
export
  x :Int
end;
```

compile *P-Quest* modules,

```
module a :A
export
  let x = 3;
end;
```

and import (link) new modules:

```
import a:A;
a.x;
import a:A;
a.x;
```

Note that the module body is evaluated only once.

5 Stability and Recovery

To preserve the top-level value, type, kind, module and interface declarations as well as the values of all transitively reachable objects including mutable values, the store has to be "stabilised". This is achieved by calling a routine from the standard module *store*.

```
import store :Store;
store.stabilise();
```

This operation can be embedded also into application program to checkpoint the store (including active stack frames) at arbitrary points in time. However, it should be noted that the stabilise operation does not preserve the state of external files or of the screen.

The operation

```
store.halt();
```

stabilises the store and immediately terminates program execution.

If you leave the interactive loop of the *P-Quest* system with CTRL-D ("^D"), or if the system crashes, all changes to the persistent store since the last checkpoint will be undone.

By invoking the *P-Quest* system with the Unix command

```
PQuestRecover
```

execution resumes with the next statement after the last *store.stabilise()* resp. *store.halt()* statement.

The following program fragment makes use of this suspension mechanism:

```
let rec stopAfter(depth :Int) :Ok =
  if depth is 0 then
    store.halt()
  else
    print.string("Entering, depth = " <> fmt.int(depth) <> "\n")
    stopAfter(depth-1)
    print.string("Leaving, depth = " <> fmt.int(depth) <> "\n")
  end;
```

```
stopAfter(5);
..let stopAfter : All(depth:Int) Ok = <fun stopAfter()>
```

```
Entering, depth = 5
Entering, depth = 4
Entering, depth = 3
Entering, depth = 2
Entering, depth = 1
Exception:
florian@dbis1> PQuestRecover
                                      ___
     _____    _____ __ __ _____ _____|   |_
    |     |__|     |  |  |  _  |   __|    _|
    |  |  |__|  |  |     |   __|__   |   |
    |   __|  |__   |_____|_____|_____|___|
    |__|        |__|

    Hamburg University
    Tycoon Language Environment Release 1.1

    Restart from last checkpoint ...

Leaving, depth = 1
Leaving, depth = 2
Leaving, depth = 3
Leaving, depth = 4
Leaving, depth = 5
```

6 Garbage Collection

The persistence model of *P-Quest* is based on transitive reachability from the "main program". Typically, this main program is the interactive compiler environment (see Sec ?? how to load a new main program into a persistent store). By introducing bindings at the top-level, new, user-defined data structures can be made reachable. There is no (unsafe) explicit object deletion mechanism. Garbage collection is either invoked explicitly

```
store.garbageCollect();
```

or implicitly, as soon as the system runs out of (persistent) memory. For example, the following program fragment triggers a garbage collection:

```
let createGarbage(count :Int) :Ok =
  (* create count arrays with 10000 integer elements *)
  for i= 1 upto count do
    let dummy = arrayOp.new(10000 0)
 end;

createGarbage(100);  (* allocate 4.000.000 bytes *)
createGarbage(100);  (* allocate 4.000.000 bytes *)
```

There is also a possibility to garbage collect a *P-Quest* store with a Unix command:

```
PQCollect .
```

This command performs a full garbage collection on the persistent store in the current directory.

7 Exchanging Data and Defining Search Paths

As you may have noticed by now, compiled interfaces and modules are created as separate Unix files with the extension ".x". Only when a module or interface is imported, this file is linked into the persistent store. This makes it possible to share libraries between separate persistent stores.

The search path for ".x" files and files loaded with the **load** command is initialized to

```
.:/:/usr/local/lib/quest.p-sparc
```

Therefore, *P-Quest* first scans the current directory, then the root directory and finally the *P-Quest* installation directory. The inclusion of the root directory allows the specification of files by their absolute path names, e.g.,

```
load "/users/dbis1/Test.qst";
```

The search path can be changed at the *P-Quest* top level as follows:

```
command "SetPath .:./graphicenv";
```

Using the generic import and export mechanisms of *P-Quest*, it is also possible to write arbitrary complex data structures (including functions) onto a file and to re-load them into another persistent store, preserving circularities and sharing within the data structure.

```
let w = writer.file("data.x");
let a = 3;
dynamic.extern(w dynamic.new(a) dynamic.portable);
writer.close(w);
```

The file *data.x* contains a single integer (a). It can be re-imported as follows:

```
let r = reader.file("data.x");
let a = dynamic.be(:Int dynamic.intern(r dynamic.portable));
reader.close(r);
a;
```

8 Linking Stand-Alone Applictions

For some applications it is desirable to have a persistent store that only contains the application program and not the full *P-Quest* compilation environment.

The following *P-Quest* declaration defines a main program function *prog* that is statically linked to the standard modules (e.g. *print, store*) it transitively imports. It is important that the main program function does not return but terminates by raising an exception:

```
let prog() :Ok =
 begin
  print("Hello World!\n")
  store.stabilise()
  print("Hello World 2!\n")
  raise exception exit:Ok end end
 end;
```

To generate a stand-alone *P-Quest* boot file in portable format, the generic export mechanism is used:

```
let w = writer.file("Prog.qm");
dynamic.extern(w dynamic.new(prog) dynamic.portable);
writer.close(w);
```

The program *Prog.qm* is then executed in the persistent store found in the current directory:

```
PQuest . prog.qm
```

Subsequent *PQuestRecover* and *PQuest* commands will execute *Prog.qm*. To re-load the compiler environment into the store, use the command:

```
PQuest . NewQuest.qm
```

9 Compatibility Issues

Since *P-Quest* uses the same portable data format like Quest Version 12, it is possible to exchange *binary data* freely between both systems. To "intern" the data file *data.x* and the program *Prog.qm* into the non-persistent Quest environment, simply write:

```
let r = reader.file("data.x");
let a = dynamic.be(:Int dynamic.internPortable(r));
reader.close(r);
a;
let r = reader.file("Prog.qm");
let f = dynamic.be(:All()Ok dynamic.internPortable(r));
reader.close(r);
f();
```

The "intern" of binary data "externed" by Quest into a *P-Quest* object store works analogously.

Restrictions: *P-Quest* and Quest Programs that make use of the *store* module as well as *P-Quest* programs that utilize the module *ccall* cannot be exchanged (they will fail at run time).

A Files Included in the Distribution Tape

File Name:	Short Description
*.spec	Source code for Tycoon library modules
*.spec.x	Compiled interface descriptions
*.impl.x	Compiled Tycoon library modules
*.info.x	Debugger information for Tycoon library modules (unused)
Install.p-sparc	Installation script (csh)
Library.qst	*P-Quest* statements executed whenever the interactive environment is entered (imports standard Tycoon modules)
NewQuest.qm	Byte code for the bootstrapped *P-Quest* system
PQCollect	Stand-alone garbage collection program
PQFormat	Program to create a new persistent store
PQInit	Program to initialize an existing persistent store
PQM	Interpreter for *P-Quest* byte code
PQStatistics	Stand-alone program to generate store statistics

References

[BR91] A.L. Brown and J. Rosenberg. Persistent Object Stores: An Implementation Technique. In *Proceedings of the Fourth International Workshop on Persis-*

tent Object Systems, Martha's Vineyard, Massachusetts. Morgan Kaufmann Publishers, January 1991.

[Car89] L. Cardelli. Typeful Programming. Digital Systems Research Center Reports 45, DEC SRC Palo Alto, May 1989.

[Car90] L. Cardelli. The Quest Language and System (Tracking Draft). Digital systems research center, DEC SRC Palo Alto, 1990. (shipped as part of the Quest V.12 system distribution).

[Mat91] F. Matthes. Tycoon Library Manual (*P-Quest* Version). DBIS Tycoon Report 102-91, Fachbereich Informatik, Universität Hamburg, West Germany, October 1991.

[MM91] B. Mathiske and F. Matthes. The P-Quest C-Call Facility. DBIS Tycoon Report 103-91, Fachbereich Informatik, Universität Hamburg, West Germany, October 1991.