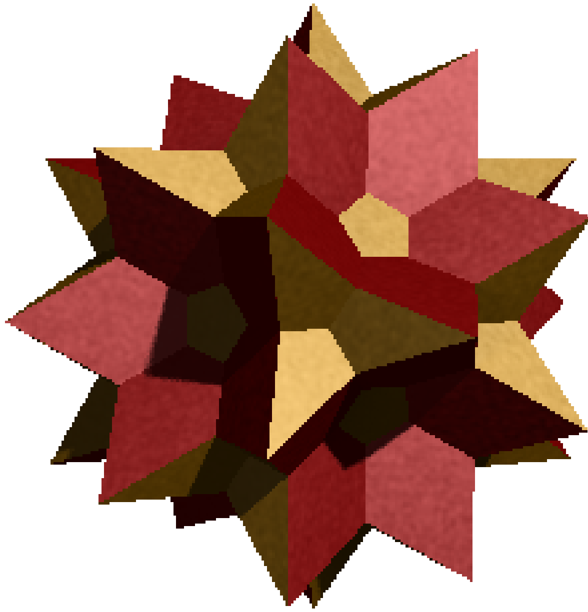
WOLFRAM TECHNOLOGY CONFERENCE 20

Queues: an Exercise in Data Type Design



Dr. Roman E. Maeder
MathConsult Dr. R. Mäder
Samstagernstraße 58a
CH−8832 Wollerau

mailto:maeder@mathconsult.ch
http://www.mathconsult.ch

Part of *Parallel Computing Toolkit* is an extensible queueing system. We discuss the design of the interface and reference implementations of different kinds of queues in an object−oriented way.

- The use of interfaces
- Queue data type requirements
- Interface design
- Implementations
- Comparison

Pluggable Data Types

Goal: Users of *Parallel Computing Toolkit* (PCT) should be able to specify a data type of their choice for the process queue. The PCT has certain requirements for possible data types and any type actually used must satisfy these requirements. How it does that is no longer a concern of PCT.

These considerations lead naturally to the definition of an *interface* between the user of a data type (PCT) and its implementor.

Let me show you how to formalize such an interface and make good use of *Mathematica*'s programming features for a clean design and easy implementation.

Interfaces

In C++ interfaces take the form of an *abstract base class*, Java uses similar ideas (and they are actually called “interfaces”). In *Mathematica* we do not have to be dogmatic about the limitations imposed on abstract classes or interfaces. We will end up with an *almost* abstract base class.

Use of Queues

PCT uses a queue to manage processes created with the `Queue[ ]` command.

```
In[3]:= ids = {Queue[1 + 1], Queue[2 + 2], Queue[3 + 3]}

Out[3]= {pid1, pid2, pid3}

In[4]:= Wait[ids]

Out[4]= {2, 4, 6}
```

The order in which processes are sent to available remote processors is determined either by the order in which they were queued or it may depend on properties of the processes, such as their intended priorities.

```
In[5]:= ids = Function[n,
            Queue[First[Timing[N[Pi, 10^n]]], Scheduling → n]] /@ {2, 3, 4, 5}

Out[5]= {pid4, pid5, pid6, pid7}

In[6]:= Wait[ids]

Out[6]= {2.77827×10^-18 Second, 2.77827×10^-18 Second, 0.05 Second, 1.66 Second}
```

Requirements

In particular, PCT requires the following operations on the process queue.

- obtain a fresh, empty queue *of a particular type*
- enter an item into the queue
- test whether the queue is empty
- retrieve and remove the *largest* element from the queue

Nice to have:

- obtain a list of the items currently in the queue
- count the number of elements in the queue
- output formatting
- a type predicate

Generalization

Allow to work with several queues. That is, the queue created is not some hidden implicit object, but a value that can be assigned to variables, passed as paramter, and so on.

A queue is a *data type:* it has in (internal) representation and methods to access its components and modify them.

Mutable or Not?

In *Mathematica*, as in Lisp, it is easier to design an *immutable* data type. Any operation that wants to modify a queue, must build and return a new queue.

```
q1 = enQueue[q0, item]
```

`q0` still contains the old queue.

Easier to use are *mutable* data types. Operations may change the internals of a data object and all variables that refer to a given element automatically see the modified element.

```
q1 = q;
enQueue[q, item]
```

Both `q` and `q1` now point to the new queue.

We will implement mutable queues. For mutable data types, there usually are two additional technical requirements:

- make a copy of a queue
- free memory associated with a queue that is no longer needed

Interface Design

In *Mathematica*, an interface takes the form of a package.

Properties of an interface package:

- no implementation part (`Begin["‘Private‘"]` … `End[]`)
- no declaration of the constructor

Parallel‘Queue‘Interface‘

Here is the first draft of the queue interface.

Preamble

```
BeginPackage["Parallel‘Queue‘Interface‘"]
```

Declaration of essential methods

```
EnQueue::usage = "EnQueue[q, item] inserts item into the queue q."

DeQueue::usage = "DeQueue[q] removes the largest item from the queue q.
    It returns the item removed."

EmptyQ::usage = "EmptyQ[q] is True if the queue q is empty."
```

More methods

```
Top::usage = "Top[q] gives the largest item in the queue q."

Size::usage = "Size[q] gives the number of elements in the queue q."

Normal::usage =
 Normal::usage <> " Normal[q] gives a list of the elements of the queue q."
```

```
Copy::usage = "Copy[q] makes a copy of the queue q."
```

```
delete::usage = "delete[q] frees storage associated with the queue q."
```

The type predicate

```
qQ::usage = "qQ[queue] is True, if queue
   is a valid queue (a subtype of this interface)."
```

That's it!

```
Protect[EnQueue, DeQueue, EmptyQ, Size, delete, Copy, qQ]
```

```
EndPackage[]
```

Where is the Constructor?

All operations declared in the interface will work with any queue implementation we may develop. The exception is the constructor: there we have to make up our mind about which queue type we want to choose. Therefore, the constructor does not belong into the interface.

The Type Predicate

How do we check whether an argument passed to one of the methods is, in fact, a proper queue? The normal way to enforce type checks in *Mathematica* is

```
method[x_type] := …
```

to restrict data to expression with a certain head (or type).

However, each of our actual queue data types will use its own type of expressions, so this does not work (*Mathematica* is not fully object–oriented, so inheritance of types is not available). We can use a predicate instead, and change the syntax only slightly:

```
method[x_?typeQ] := …
```

Next Steps

- To implement a queue, extend the interface package, declare the constructor and implement all methods.
- To use a queue, import one of the implementation packages and use its constructor and the methods declared in the interface (which will be imported as well).

However, we can do some of the implementation work already in the interface, so let us dwell on it some more before we dig into the implementations.

Generic Operations

Some of the auxiliary methods declared in the interface can be implemented in a way that is completely independent of the actual queue data type.

```
Normal[q0_?qQ] :=
 Module[{li = {}, q = Copy[q0]},
  While[! EmptyQ[q], AppendTo[li, DeQueue[q]]];
  delete[q];
  li
 ]
```

```
Size[q_?qQ] := Length[Normal[q]]
```

```
EmptyQ[q_?qQ] := Size[q] == 0
```

```
Top[q0_?qQ] :=
 Module[{res, q = Copy[q0]}, res = DeQueue[q];
  delete[q];
  res
 ]
```

and also formatting, which we skip in this presentation.

Some of these are obviously inefficient.

RecursionLimit?

Aren't these implementations circular (`EmptyQ`—`Normal`—`Size`)?

Yes they are! Each implementation will have to *override* at least one of them. We can choose which one; there is usually one method that has a particularly efficient implementation, depending on the way we implement the queue in question.

Pure Virtual (or Abstract, or …)

What happens, if we forget to implement one of the required methods, or call `DeQueue` with an empty queue?

We can decide right here in the interface be providing fallback definitions (that should never be used).

```
EnQueue[q_?qQ, _] := (Message[EnQueue::NIM, EnQueue, Head[q]]; $Failed)
DeQueue[q_?qQ] := (Message[DeQueue::NIM, DeQueue, Head[q]]; $Failed)
Copy[q_?qQ] := (Message[Copy::NIM, Copy, Head[q]]; $Failed)

General::NIM = "Method `1` not implemented in class `2`."

qQ[_] = False

Top[q_?EmptyQ] := (Message[DeQueue::empty, q]; $Failed)
DeQueue[q_?EmptyQ] := (Message[DeQueue::empty, q]; $Failed)
```

Finally: an Implementation

Let us now implement the first of any possible number of queue data types that satisfies the given interface. Technically, these data types are subtypes of the interface, which becomes apparent in their package declaration.

```
BeginPackage["Parallel`Queue`WAWA`","Parallel`Queue`Interface`"]
```

How to make them Mutable?

The only destructive operation in *Mathematica* is the assignment of a value to a variable. Values can be changed, and everywhere we look at a variable we see the same value.

Therefore, queue data elements will be expressions that contain a private variable (in unevaluated form) to which we can assign new value as we modify the queue.

Each data type will use its own symbol as the head of the data elements.

Quick, but not Dirty

Here are Lisp lists.

```
BeginPackage["Parallel`Queue`Lisp`", "Parallel`Queue`Interface`"]

LispQueue::usage = "LispQueue[] creates an empty queue."
```

Implementation

The items are stored as cons[*e1*,cons[*e2*,...,cons[]...]], just as in Lisp

```
SetAttributes[cons, HoldAllComplete]
`empty = cons[] (*shared nil*)
car[cons[car_, cdr_]] := car
cdr[cons[car_, cdr_]] := cdr
```

Data type head

```
SetAttributes[{queue}, HoldAll]
```

Constructors. Note how we create a new variable with a value and and insert it into the unevaluated data object.

```
LispQueue[] := Module[{r = empty}, queue[r]]

queue /: Copy[queue[s_]] := Module[{r = s}, queue[r]]
```

Methods

Note how we isolate the variable inside the data element and obtain and change its value.

```
queue /: EnQueue[q : queue[r_], val_] := (With[{rv = r}, r = cons[val, rv]]; q)

queue /: DeQueue[q : queue[r_] /; ! EmptyQ[q]] :=
 With[{res = car[r]}, r = cdr[r]; res]
```

Override generics as desired.

```
queue /: EmptyQ[queue[r_]] := r === empty

queue /: Top[q : queue[r_]] /; ! EmptyQ[q] := car[r]

queue /: delete[queue[r_]] := (r = empty;)
```

Use generic implementations for Size[], Normal[] (we could override Normal for speed).

Wrapping Up

Predicate

```
queue /: qQ[queue[r_]] := ValueQ[r]

End[]

EndPackage[]
```

Testing

```
In[1]:= Needs["Parallel`Queue`Lisp`"]
```

The construct creates a new queue.

```
In[2]:= q = LispQueue[]

Out[2]= ()
```

Here is its internal representation.

```
In[3]:= FullForm[q]

Out[3]//FullForm= Parallel`Queue`Lisp`Private`queue[Parallel`Queue`Lisp`Private`r$15]

In[4]:= List @@ q

Out[4]= {Parallel`Queue`Lisp`Private`cons[]}
```

Exercise the Methods

```
In[4]:= EnQueue[q, a]

Out[4]= (a)

In[5]:= Size[q]

Out[5]= 1

In[6]:= EnQueue[q, b]

Out[6]= (b  a)

In[7]:= EnQueue[q, c]

Out[7]= (c  b  a)

In[8]:= Normal[q]

Out[8]= {c, b, a}

In[9]:= EmptyQ[q]

Out[9]= False

In[10]:= DeQueue[q]

Out[10]= c

In[11]:= DeQueue[q]

Out[11]= b

In[12]:= DeQueue[q]

Out[12]= a

In[13]:= DeQueue[q]

   DeQueue::empty : Queue () is empty.

Out[13]= $Failed

In[14]:= EnQueue[q, #] & /@ Range[20];
         q

Out[14]= (20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1)
```

Other Implementations

- `FIFO`: first–in first–out, the default queue of PCT

  Uses a circular buffer that is expanded as needed.

- `LIFO`: last–in first–out (just like Lisp)

  uses an array that is expanded as needed

- `Priority`: a true priority queue

  uses a heap in an array

Comparisons

```
TableForm[Outer[First[#1@#2] /. Second → 1 &, tests, $QueueTypes],
 TableHeadings → {tests, $QueueTypes}]
```

	FIFOQueue	priorityQueue	LIFOQueue	LispQueue
test1	1.54	5.45	0.95	0.63
test2	1.64	5.82	0.86	0.78
test3	0.	4.42	0.01	0.69
test4	1.99	6.56	1.18	1.12

test1: enQueue, then deQueue
test2: repeated sequences of enQueue/deQueue
test3: Normal[]
test4: random positive walk

Summary

- a *Mathematica* realization of a clean data type design and implementation
- interface specification
- generic operations
- implementations as subtypes
- mutable data

The `Parallel`Queue`*` packages are bundled with PCT Version 2 in source form.